

LabVIEW™ Core 1 Course Manual

Course Software Version 2012

August 2012 Edition

Part Number 325290D-01

Copyright

© 1993–2012 National Instruments. All rights reserved.

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.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

End-User License Agreements and Third-Party Legal Notices

You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the <National Instruments>_Legal Information and <National Instruments> directories.
- EULAs are located in the <National Instruments>\Shared\MDF\Legal\License directory.
- Review <National Instruments>_Legal Information.txt for more information on including legal information in installers built with NI products.

Trademarks

CVI, LabVIEW, National Instruments, NI, ni.com, the National Instruments corporate logo, and the Eagle logo are trademarks of National Instruments Corporation. Refer to the *Trademark Information* at ni.com/trademarks for other National Instruments trademarks.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or the *National Instruments Patent Notice* at ni.com/patents.

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code *feedback*.

Contents

Student Guide

A. NI Certification	v
B. Course Description	vi
C. What You Need to Get Started	vii
D. Installing the Course Software	viii
E. Course Goals	viii
F. Course Conventions	ix

Lesson 1

Navigating LabVIEW

A. What is LabVIEW	1-2
B. Project Explorer	1-5
C. Parts of a VI	1-10
D. Front Panel	1-14
E. Block Diagram	1-20
F. Searching for Controls, VIs and Functions	1-31
G. Selecting a Tool	1-33
H. Dataflow	1-38
I. Building a Simple VI	1-39

Lesson 2

Troubleshooting and Debugging VIs

A. Correcting Broken VIs	2-2
B. Debugging Techniques	2-3
C. Undefined or Unexpected Data	2-9
D. Error Handling	2-10

Lesson 3

Implementing a VI

A. Front Panel Basics	3-2
B. LabVIEW Data Types	3-3
C. Documenting Code	3-14
D. While Loops	3-16
E. For Loops	3-19
F. Timing a VI	3-22
G. Data Feedback in Loops	3-23
H. Plotting Data – Waveform Chart	3-26
I. Case Structures	3-28

Lesson 4

Developing Modular Applications

A. Understanding Modularity	4-2
B. Building the Icon and Connector Pane	4-3
C. Using SubVIs	4-7

Lesson 5

Creating and Leveraging Data Structures

A. Arrays	5-2
B. Common Array Functions	5-5
C. Polymorphism	5-6
D. Auto-Indexing	5-8
E. Clusters	5-13
F. Type Definitions	5-18

Lesson 6

Managing File and Hardware Resources

A. Understanding Hardware and Software Resources	6-2
B. File I/O	6-4
C. Acquiring Measurements with a DAQ System	6-7
D. Controlling Instruments	6-16

Lesson 7

Using Sequential and State Machine Algorithms

A. Using Sequential Programming	7-2
B. Using State Programming	7-4
C. State Machines	7-4

Lesson 8

Solving Dataflow Challenges with Variables

A. Communicating Between Parallel Loops	8-2
B. Writing to Controls and Reading from Indicators	8-4
C. Variables	8-4
D. Race Conditions	8-13

Appendix A

Additional Information and Resources

Glossary

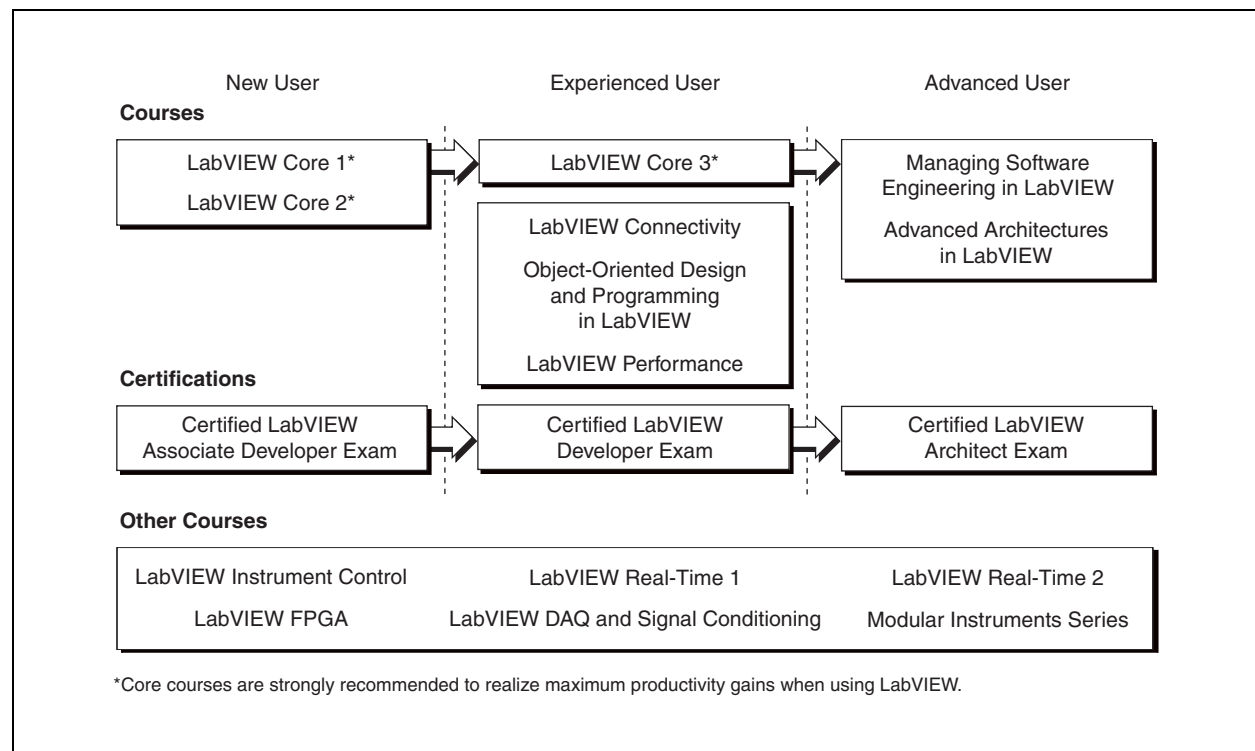
Student Guide

Thank you for purchasing the *LabVIEW Core 1* course kit. You can begin developing an application soon after you complete this course. This course manual and the accompanying software are used in the three-day, hands-on *LabVIEW Core 1* course.

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

A. NI Certification

The *LabVIEW Core 1* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for the NI Certified LabVIEW Associate Developer exam. The following illustration shows the courses that are part of the LabVIEW training series. Refer to ni.com/training for more information about NI Certification.



B. Course Description

The *LabVIEW Core 1* course teaches you programming concepts, techniques, features, VIs, and functions you can use to create test and measurement, data acquisition, instrument control, datalogging, measurement analysis, and report generation applications. This course assumes that you are familiar with Windows and that you have experience writing algorithms in the form of flowcharts or block diagrams. The course and exercise manuals are divided into lessons, described as follows.

In the course manual, each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A description of the topics in the lesson
- A summary quiz that tests and reinforces important concepts and skills taught in the lesson

In the exercise manual, each lesson consists of the following:

- A set of exercises to reinforce those topics
- Some lessons include optional and challenge exercise sections or a set of additional exercises to complete if time permits



Note For course and exercise manual updates and corrections, refer to ni.com/info and enter the Info Code `Core1`.

Several exercises use one of the following National Instruments hardware products:

- A plug-in multifunction data acquisition (DAQ) device connected to a BNC-2120 containing a temperature sensor, function generator, and LEDs
- A GPIB interface connected to an NI Instrument Simulator

If you do not have this hardware, you still can complete the exercises. Alternate instructions are provided for completing the exercises without hardware. You also can substitute other hardware for those previously mentioned. For example, you can use a GPIB instrument in place of the NI Instrument Simulator, or another National Instruments DAQ device connected to a signal source, such as a function generator.

C. What You Need to Get Started

Suggested Reading

The suggested reading material ensure that all students have a minimum knowledge of key theories and concepts related to the *LabVIEW Core 1* course. To get the most out of this course, complete all the suggested reading material.

To access each of the following suggested reading materials, refer to ni.com/info and enter the Info Code that corresponds to each topic:

- ☐ *LabVIEW Core 1 - The Software Development Method* (Info Code: SoftDev)
- ☐ *Introduction to Data Acquisition* (Info Code: DAQ)
- ☐ *GPIB Instrument Control Tutorial* (Info Code: GPIB)

Course Materials

Before you begin this course, ensure you have all the following items:

- ☐ Computer running Windows 7/Vista/XP
- ☐ Multifunction DAQ device configured as Dev1 using Measurement & Automation Explorer (MAX)
- ☐ BNC-2120, wires, and cable
- ☐ GPIB interface
- ☐ NI Instrument Simulator and power supply
- ☐ LabVIEW Full or Professional Development System 2012 or later
- ☐ DAQmx 9.5.5 or later
- ☐ NI-488.2 3.0.2 or later
- ☐ NI VISA 5.2 or later
- ☐ A GPIB cable

- ☐ NI Instrument Simulator Wizard installed from the NI Instrument Simulator software CD
- ☐ *LabVIEW Core 1* course CD, which installs the following folders:

Directory	Description
Exercises	Folder for saving VIs created during the course and for completing certain course exercises; also includes subVIs necessary for some exercises and zip file (NI Instrument Simulator.zip) containing the LabVIEW instrument driver for the NI Instrument Simulator
Solutions	Contains the solutions to all the course exercises

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer. The **LabVIEW Core 1 Course Setup** dialog box appears.
2. Click **Install the course materials**.
3. Follow the onscreen instructions to complete installation and setup.

Exercise files are located in the <Exercises>\LabVIEW Core 1\ folder.



Note Folder names in angle brackets, such as <Exercises>, refer to folders on the root directory of your computer.

E. Course Goals

This course prepares you to do the following:

- Understand front panels, block diagrams, icons, and connector panes
- Use the programming structures and data types that exist in LabVIEW
- Use various editing and debugging techniques
- Create and save VIs so you can use them as subVIs
- Display and log data
- Create applications that use plug-in DAQ devices
- Create applications that use serial port and GPIB instruments




This course does *not* describe the following:

- Every built-in VI, function, or object; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Analog-to-digital (A/D) theory

- Operation of the GPIB bus
- Developing an instrument driver
- Developing a complete application for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create

F. Course Conventions

The following conventions appear in this course manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence Tools»Instrumentation»Find Instrument Drivers directs you to drop down the Tools menu, select the Instrumentation item, and finally select the Find Instrument Drivers option.
	This icon denotes a tip, which alerts you to advisory information.
	This icon denotes a note, which alerts you to important information.
	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes sections of dialog boxes and hardware labels.
<i>italic</i>	Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.
<code>monospace</code>	Text in this font denotes text or characters that you enter from the keyboard, sections of code, programming examples, and syntax examples. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
<code>monospace bold</code>	Text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Navigating LabVIEW

This lesson introduces how to navigate the LabVIEW environment. This includes using the menus, toolbars, palettes, tools, help, and common dialog boxes of LabVIEW. You also learn how to run a VI and gain a general understanding of a front panel and block diagram. At the end of this lesson, you create a simple VI that acquires, analyzes, and presents data.

Topics

- A. What is LabVIEW
- B. Project Explorer
- C. Parts of a VI
- D. Front Panel
- E. Block Diagram
- F. Searching for Controls, VIs and Functions
- G. Selecting a Tool
- H. Dataflow
- I. Building a Simple VI

A. What is LabVIEW

LabVIEW is a graphical programming environment you can use to quickly and efficiently create applications with professional user interfaces. Millions of engineers and scientists use LabVIEW to develop sophisticated measurement, test, and control system applications using intuitive icons and wires. In addition, the LabVIEW platform is scalable across different targets and OSs. In fact, LabVIEW offers unrivaled integration with thousands of hardware devices and provides hundreds of built-in libraries for advanced analysis and data visualization for you to create virtual instruments you can customize to your needs.

Because LabVIEW programs imitate the appearance and operation of physical instruments, such as oscilloscopes and multimeters, LabVIEW programs are called virtual instruments or, more commonly, VIs. VIs have front panels and block diagrams. The front panel is the user interface. The block diagram is the programming behind the user interface. After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The code on the block diagram is graphical code, also known as G code or block diagram code.

In contrast to text-based programming languages, like C++ and Visual Basic, LabVIEW uses icons instead of lines of text to create applications. In text-based programming, instructions determine the order of program execution. LabVIEW uses graphical dataflow programming. In graphical dataflow programming, the flow of data through the nodes on the block diagram determines the execution order. Graphical programming and dataflow execution are the two major ways LabVIEW is different from most other general-purpose programming languages.

In this course, you see how you can use LabVIEW to effectively create simple data acquisition applications using the three steps: acquire, analyze, and present. Although this course is taught on a Windows system, LabVIEW is multi-platform. You can develop applications on a Windows, Mac OS, or Linux system. Furthermore, you can deploy LabVIEW applications to a variety of real-time and FPGA targets.

LabVIEW Characteristics

LabVIEW programs have the following characteristics:

- A graphical and compiled nature
- Dataflow and/or event-based programming
- Multi-target and platform capabilities
- Object-oriented flexibility
- Multi-threading possibilities

In LabVIEW Core 1, you learn about the graphical and compiled nature of LabVIEW and dataflow programming. The following courses explore other characteristics:

- LabVIEW Core 2—Learn about event-based programming.
- Object-Oriented Design and Programming in LabVIEW—Learn about object-oriented programming in LabVIEW.
- LabVIEW Performance—Learn how to take advantage of LabVIEW's multithreading and memory management to improve program execution and memory usage.

Graphical and Compiled

While represented graphically, with icons and wires instead of with text, G code on the block diagram contains the same programming concepts found in most traditional languages. For example, G code includes data types, loops, event handling, variables, recursion, and object-oriented programming. LabVIEW compiles G code directly to machine code so the computer processors can execute it. You do not have to compile G code in a separate step.

Dataflow and Event-Driven Programming

LabVIEW programs execute according to dataflow programming rules instead of the procedural approach found in most text-based programming languages such as C and C++. Dataflow execution is data-driven, or data-dependent. The flow of data between nodes in the G code determines the execution order.

Event-driven programming features extend the LabVIEW dataflow environment to allow the user's direct interaction with the program without the need for polling. Event-based programming also allows other asynchronous activity to influence the execution of G code on the block diagram.

Multi-Target and Multi-Platform

With LabVIEW applications, you can target multicore processors and other parallel hardware such as field-programmable gate arrays (FPGAs). You can automatically scale LabVIEW applications to CPUs with two, four, or more cores, often with no additional programming effort.

G code, with the exception of a few platform-specific functions, is portable between the different LabVIEW systems for different operating systems. Therefore, you can often use the same code whether running LabVIEW on Windows, Mac OS X or Linux systems.

Object-Oriented

Object-oriented programming is a popular programming approach across a wide variety of programming languages. It allows a variety of similar, yet different items, to be represented as a class of objects in software. LabVIEW provides tools and functions so you can use object-oriented programming techniques in G code.

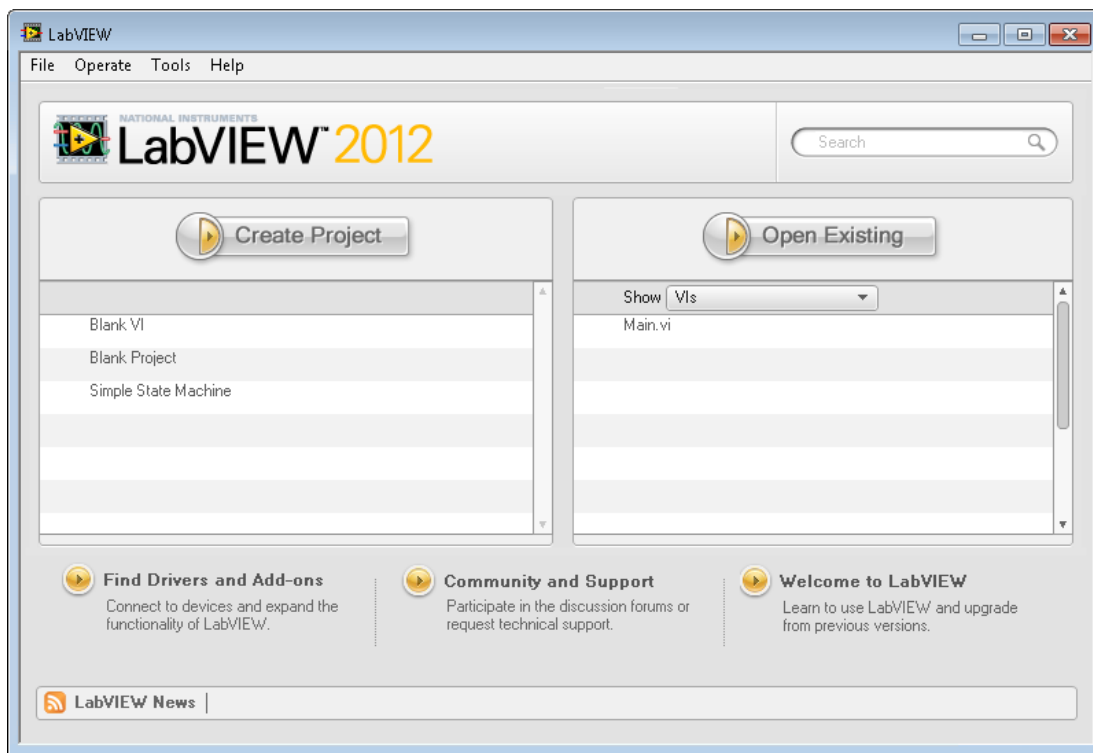
Multithreading and Memory Management

LabVIEW enables your code to have automatic parallelism. In other languages if you want to run code in parallel, you have to manage multiple threads manually. The LabVIEW environment, with the compiler and execution system working together, automatically runs code in parallel whenever possible. Most of the time the details of the execution system are unimportant to you because the system does the right thing without intervention. However, LabVIEW also provides you with options for improving performance.

Launching the LabVIEW Environment

When you launch LabVIEW, the **Getting Started** window appears as shown in Figure 1-1.

Figure 1-1. LabVIEW Getting Started Window



Use the **Getting Started** window to create new projects and VIs. You can create items from scratch or from templates and samples. You can also open existing LabVIEW files and access LabVIEW community resources and help.

The **Getting Started** window disappears when you open an existing file or create a new file, and reappears when you close all open front panels and block diagrams. You can display the window at any time by selecting **View»Getting Started Window**.

B. Project Explorer

VIs are LabVIEW programs, and you can use multiple VIs together to make a LabVIEW application. To group these application-related VIs together, use a LabVIEW project. When you save a LabVIEW project from the **Project Explorer** window, LabVIEW creates a project file (.lvproj), that includes references to all the LabVIEW files and non-LabVIEW files in the project, configuration information, build information, and deployment information.

Some LabVIEW applications, such as simple VIs, do not need to use a LabVIEW project. However, you must use a project to build stand-alone applications and shared libraries. In addition, you must use a project to work with non-development machine targets such as real-time (RT), field-programmable gate array (FPGA), or personal digital assistant (PDA) targets. Refer to the specific module documentation for more information about using projects with the LabVIEW Real-Time, FPGA, and PDA modules.

Project Explorer Window

Projects in LabVIEW consist of VIs, files necessary for those VIs to run properly, and supplemental files such as documentation or related links. Use the **Project Explorer** window to manage projects in LabVIEW.

The **Project Explorer** window includes the following items by default:

- **Project root**—Contains all other items in the **Project Explorer** window. The label on the project root includes the filename for the project.
- **My Computer**—Represents the local computer as a target in the project.
- **Dependencies**—Includes VIs and items that VIs under a target require.
- **Build Specifications**—Includes build configurations for source distributions and other types of builds available in LabVIEW toolkits and modules. If you have the LabVIEW Professional Development System or Application Builder installed, you can use **Build Specifications** to configure stand-alone applications, shared libraries, installers, and zip files.



Tip A *target* is any device that can run a VI.

When you add another target to the project, LabVIEW creates an additional item in the **Project Explorer** window to represent the target. Each target also includes **Dependencies** and **Build Specifications** sections. You can add files under each target.

Complete the following steps to use the **Project Explorer** window to create and edit LabVIEW Projects.

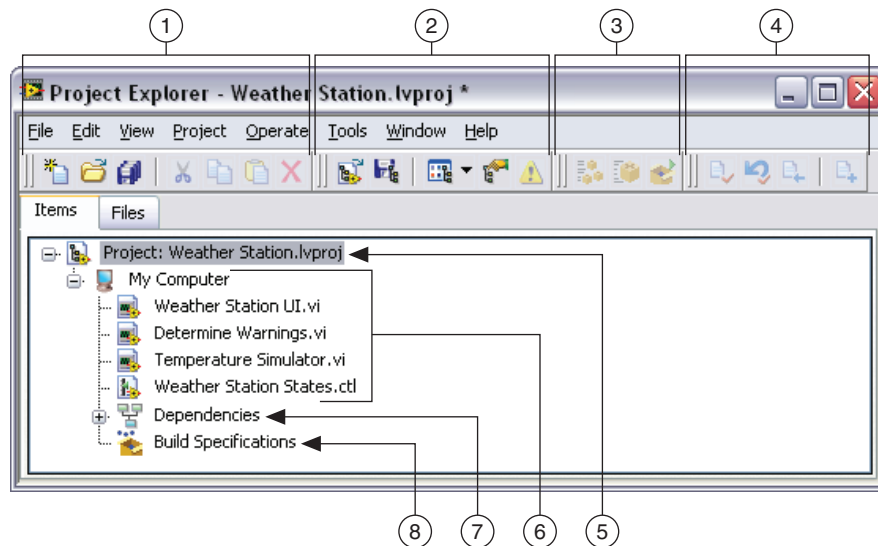
1. Select **File»Create Project** to display the **Create Project** dialog box. The **Create Project** dialog box includes a list of templates and sample projects you can use to ensure that the project you create uses reliable designs and programming practices.

2. Search for the template or sample project that supports the goal you want the project to accomplish. Use the following features to search for templates and sample projects:
 - **Filters**—Displays only results of a certain type, such as templates or sample projects for a specific target.
 - **Additional Search**—Searches the keywords, titles, and descriptions of the filtered results.
 - **More Information**—Opens the help file for the item. Review the help file to ensure that the template or sample project supports the goal you want the project to accomplish.
3. Click the **Next** or **Finish** button to configure details of the project, including how the project is saved. After you finish configuring the project, LabVIEW saves the project and opens the **Project Explorer** window and the top-level VI of the project.
4. Use the **Project Explorer** window to modify the project. Refer to the comments on the block diagrams of VIs in the project for more information about how to modify the project. Also refer to the **Project Documentation** folder in the **Project Explorer** window for more information about how to modify the project.

Project-Related Toolbars

Use the **Standard**, **Project**, **Build**, and **Source Control** toolbar buttons to perform operations in a LabVIEW project. The toolbars are available at the top of the **Project Explorer** window, as shown in Figure 1-2. You might need to expand the **Project Explorer** window to view all of the toolbars.

Figure 1-2. Project Explorer Window



- | | |
|--------------------------|------------------------|
| 1 Standard Toolbar | 5 Project Root |
| 2 Project Toolbar | 6 Target |
| 3 Build Toolbar | 7 Dependencies |
| 4 Source Control Toolbar | 8 Build Specifications |



Tip The **Source Control** toolbar is only available if you have source control configured in LabVIEW.

You can show or hide toolbars by selecting **View»Toolbars** and selecting the toolbars you want to show or hide. You can also right-click an open area on the toolbar and select the toolbars you want to show or hide.

Creating a LabVIEW Project

Complete the following steps to create a project.

1. Choose one of the following actions to create a new LabVIEW project.
 - In the **Getting Started** window, click **Blank Project** to open a blank project or click the **Create Project** button to create a project from a template.
 - In an open LabVIEW project or VI, select **File»Create Project**.
 - In any LabVIEW window, select **File»New**. Then select **Project»Empty Project** in the **New** dialog box to display the **Project Explorer** window.
2. Add items you want to include in the project under a target.
3. Select **File»Save Project** to save the project.

Adding Existing Files To A Project

You can add existing files to a project. Use the **My Computer** item (or other target) in the **Project Explorer** window to add files such as VIs or text files, to a LabVIEW project.

You can add items to a project in the following ways:

- Right-click **My Computer** and select **Add»File** from the shortcut menu to add a file. You also can select **Project»Add To Project»File** from the **Project Explorer** menu to add a file.
- Right-click the target and select **Add»Folder (Auto-populating)** from the shortcut menu to add an auto-populating folder. You also can select **Project»Add To Project»Add Folder (Auto-populating)** to add an auto-populating folder. LabVIEW continuously monitors and updates the folder according to changes made in the project and on disk.
- Right-click the target or **My Computer** and select **Add»Folder (Snapshot)** from the shortcut menu to add a virtual folder. You also can select **Project»Add To Project»Add Folder (Snapshot)** to add a virtual folder. When you select a directory on disk, LabVIEW creates a new virtual folder in the project with the same name as the directory on disk. LabVIEW also creates project items that represent the contents of the entire directory, including files and contents of subdirectories. Selecting a folder on disk adds contents of the entire folder, including files and contents of subfolders.



Note After you add a virtual folder on disk to a project, LabVIEW does not automatically update the folder in the project if you make changes to the folder on disk.

- Right-click the target and select **New»VI** from the shortcut menu to add a new, blank VI. You also can select **File»New VI** or **Project»Add To Project»New VI** to add a new, blank VI.

- Select the VI icon in the upper right corner of a front panel or block diagram window and drag the icon to the target.
- Select an item or folder from the file system on your computer and drag it to the target.

Removing Items from a Project

You can remove items from the **Project Explorer** window in the following ways:

- Right-click the item you want to remove and select **Remove from Project** from the shortcut menu.
- Select the item you want to remove and press <Delete>.
- Select the item you want to remove and click the **Remove From Project** button on the **Standard** toolbar.



Note Removing an item from a project does not delete the item on disk.

Organizing Items in a Project

The **Project Explorer** window includes two pages, the **Items** page and the **Files** page. The **Items** page displays the project items as they exist in the project tree. The **Files** page displays the project items that have a corresponding file on disk. You can organize filenames and folders on this page. Project operations on the **Files** page both reflect and update the contents on disk. You can switch from one page to the other by right-clicking a folder or item under a target and selecting **Show in Items View** or **Show in Files View** from the shortcut menu.

Use folders to organize items in the **Project Explorer** window. You can add two types of folders to a LabVIEW project, virtual folders and auto-populating folders. Virtual folders organize project items. Right-click a target in the **Project Explorer** window and select **New»Virtual Folder** from the shortcut menu to create a new virtual folder. Auto-populating folders update in real time to reflect the contents of folders on disk. Add an auto-populating folder to the project to view project items as they appear on disk.

Auto-populating folders are visible only on the **Items** page of the **Project Explorer** window. You can view the disk contents of an auto-populating folder but you cannot perform disk operations such as renaming, reorganizing, and removing project items. To perform disk operations of items in an auto-populating folder, use the **Files** page of the **Project Explorer** window. The **Files** page displays the location of project folders on disk. Project operations on the **Files** page both update and reflect the contents of the folder on disk. Likewise, LabVIEW automatically updates the auto-populating folder in the project if you make changes to the folder on disk outside of LabVIEW.

You can arrange items in a folder. Right-click a folder and select **Arrange By»Name** from the shortcut menu to arrange items in alphabetical order. Right-click a folder and select **Arrange By»Type** from the shortcut menu to arrange items by file type.

Viewing Files in a Project

When you add a file to a LabVIEW project, LabVIEW includes a reference to the file on disk. Right-click a file in the **Project Explorer** window and select **Open** from the shortcut menu to open the file in its default editor.

Right-click the project and select **View»Full Paths** from the shortcut menu to view where files that a project references are saved on disk.

Use the **Project File Information** dialog box to view where files that a project references are located on disk and in the **Project Explorer** window. Select **Project»File Information** to display the **Project File Information** dialog box. You also can right-click the project and select **View»File Information** from the shortcut menu to display the **Project File Information** dialog box.

Saving a Project

You can save a LabVIEW project in the following ways:

- Select **File»Save Project**.
- Select **Project»Save Project**.
- Right-click the project and select **Save** from the shortcut menu.
- Click the **Save Project** button on the **Project** toolbar.

You must save new, unsaved files in a project before you can save the project. When you save a project, LabVIEW does not save dependencies as part of the project file.



Note Make a backup copy of a project when you prepare to make major revisions to the project.

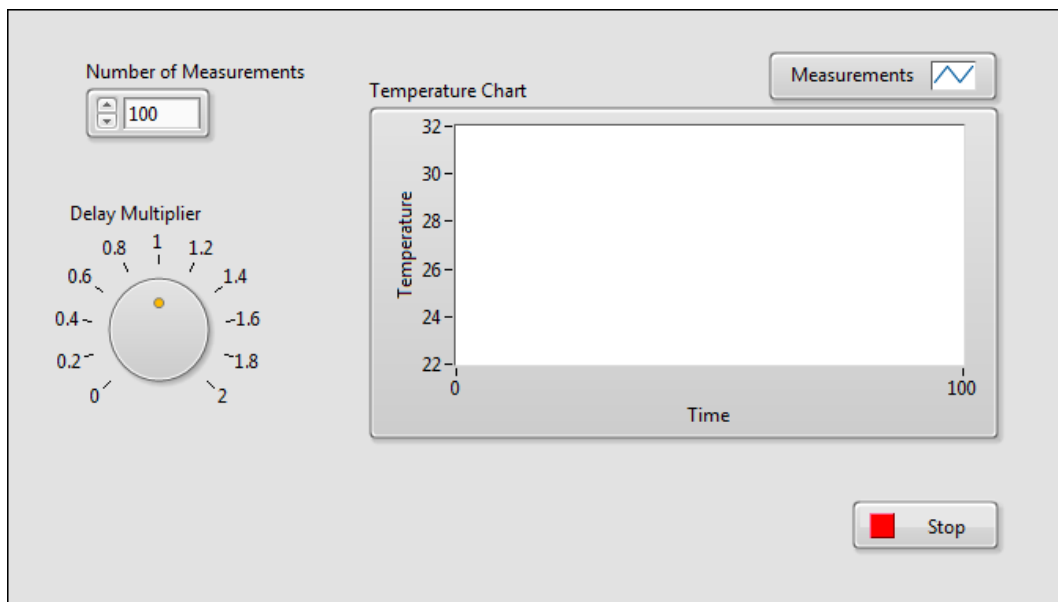
C. Parts of a VI

LabVIEW VIs contain three main components—the front panel window, the block diagram, and the icon/connector pane.

Front Panel Window

The front panel window is the user interface for the VI. Figure 1-3 shows an example of a front panel window. You create the front panel window with controls and indicators, which are the interactive input and output terminals of the VI, respectively.

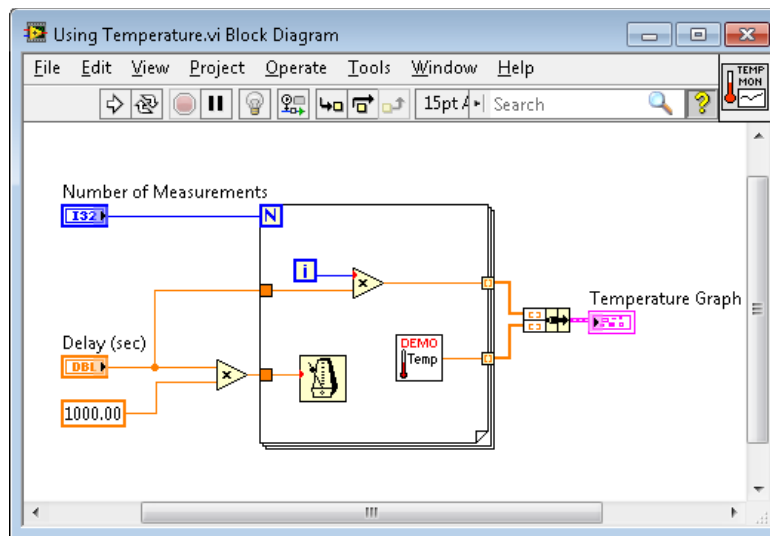
Figure 1-3. VI Front Panel



Block Diagram Window

After you create the front panel window, you add code using graphical representations of functions to control the front panel objects. Figure 1-4 shows an example of a block diagram window. The block diagram window contains this graphical source code. Front panel objects appear as terminals on the block diagram.

Figure 1-4. Block Diagram



Icon and Connector Pane

The Icon and Connector pane allows you to use and view a VI in another VI. A VI that is used in another VI is called a subVI, which is similar to a function in a text-based programming language. To use a VI as a subVI, it must have an icon and a connector pane.

Every VI displays an icon in the upper right corner of the front panel window and block diagram window. An example of the default icon is shown below. An icon is a graphical representation of a VI. The icon can contain both text and images. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. The default icon contains a number that indicates how many new VIs you opened after launching LabVIEW.



To use a VI as a subVI, you need to build a connector pane, shown below. The connector pane is a set of terminals on the icon that corresponds to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane is displayed next to the VI icon in the upper right corner of the front panel window. You cannot access the connector pane from the icon in the block diagram window.



Opening an Existing VI

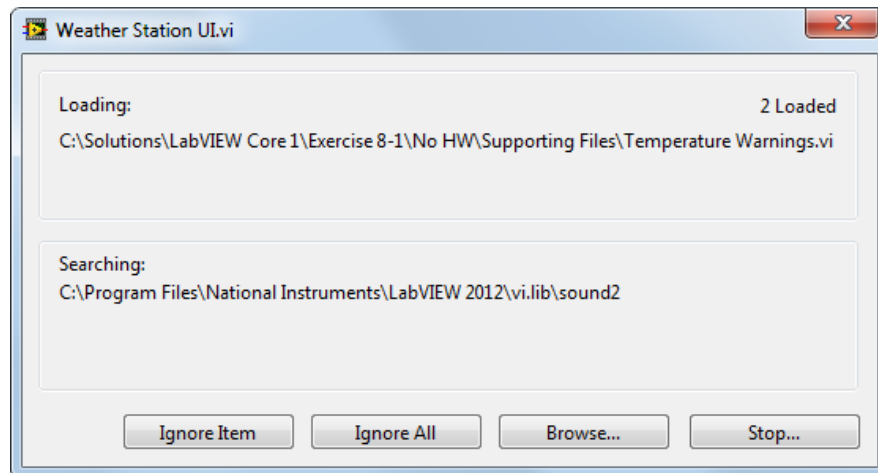
Select **Browse** in the **Open** list in the **Getting Started** window to navigate to and open an existing VI.



Tip The VIs you edit in this course are located in the <Exercises>\LabVIEW Core 1 directory.

As the VI loads, a status dialog box similar to the following example might appear.

Figure 1-5. Dialog Box Indicating the Status of Loading VIs



The **Loading** section lists the subVIs of the VI as they load into memory and shows the number of subVIs loaded into memory so far. You can cancel the load at any time by clicking the **Stop** button.

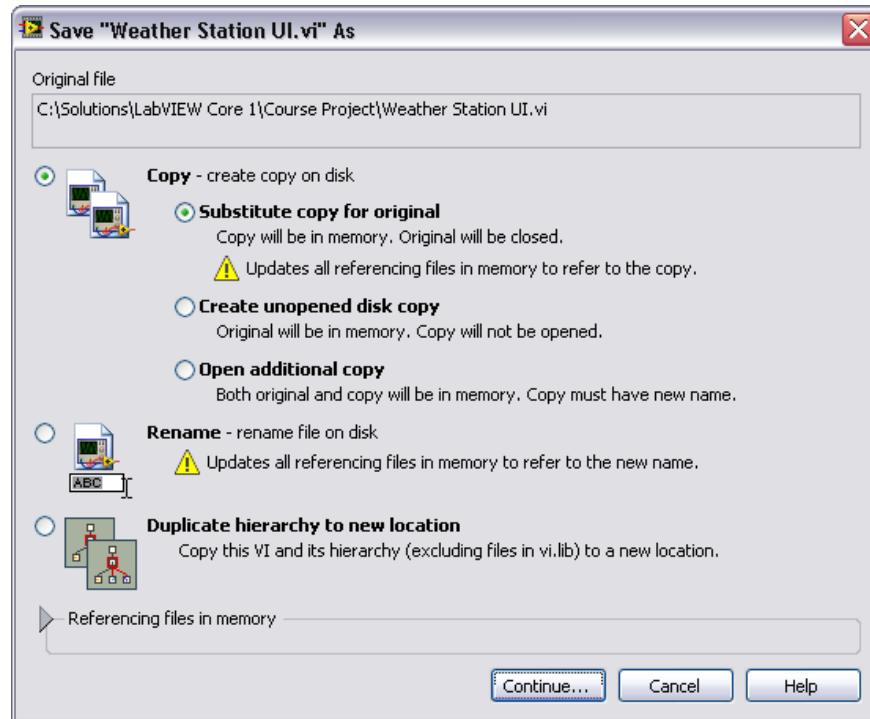
If LabVIEW cannot immediately locate a subVI, it begins searching through all directories specified by the VI search path. You can edit the VI search path by selecting **Tools»Options** and selecting **Paths** from the **Category** list.

You can have LabVIEW ignore a subVI by clicking the **Ignore Item** button, or you can click the **Browse** button to search for the missing subVI.

Saving a VI

To save a new VI, select **File»Save**. If you already saved your VI, select **File»Save As** to access the **Save As** dialog box. From the **Save As** dialog box, you can create a copy of the VI, or delete the original VI and replace it with the new one.

Figure 1-6. Save As Dialog Box

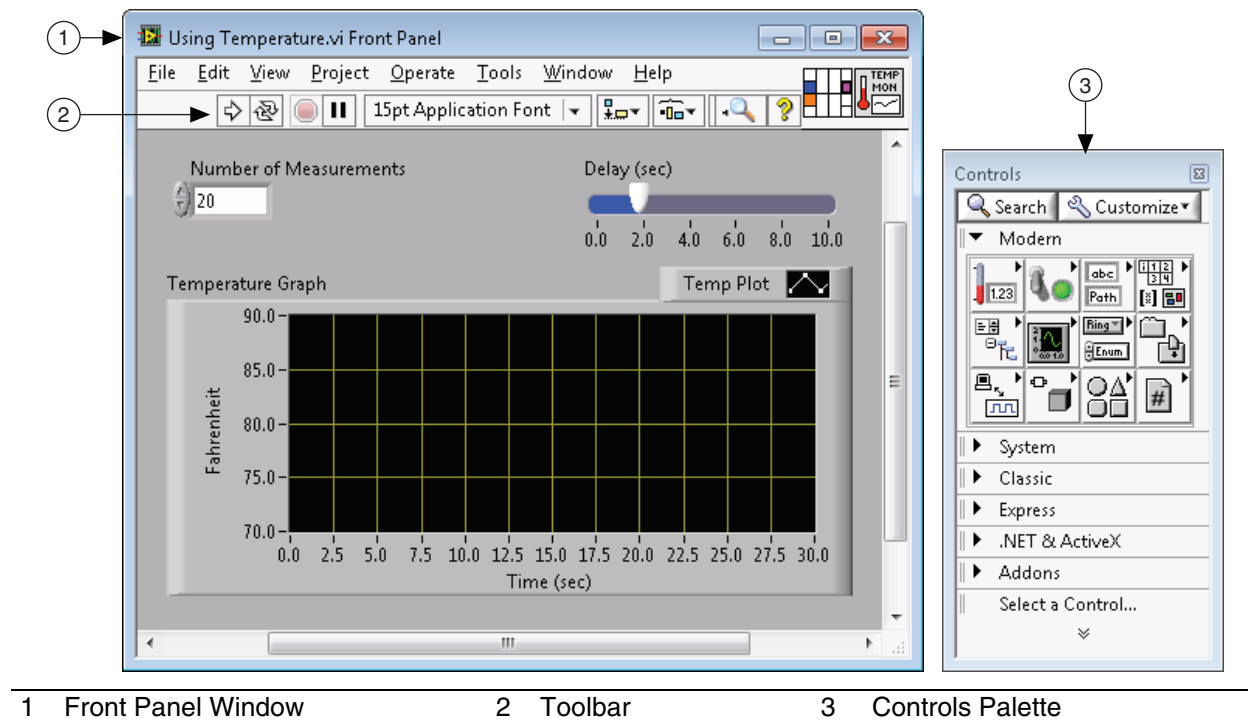


Note Refer to the *Save As Dialog Box* topic of the *LabVIEW Help* for detailed information about each option in the Save As dialog box.

D. Front Panel

When you open a new or existing VI, the front panel window of the VI appears. The front panel window is the user interface for the VI. Figure 1-7 shows an example of a front panel window.

Figure 1-7. Example of a Front Panel



Controls and Indicators

You create the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates.

Figure 1-7 has the following objects: two controls: **Number of Measurements** and **Delay(sec)**. It has one indicator: an XY graph named **Temperature Graph**.

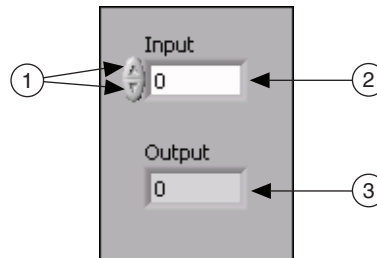
The user can change the input value for the **Number of Measurements** and **Delay(sec)** controls. The user can see the value generated by the VI on the **Temperature Graph** indicator. The VI generates the values for the indicators based on the code created on the block diagram. You learn about this in the *Numeric Controls and Indicators* section.

Every control or indicator has a data type associated with it. For example, the **Delay(sec)** horizontal slide is a numeric data type. The most commonly used data types are numeric, Boolean value and string. You learn about other data types in Lesson 3, *Implementing a VI*.

Numeric Controls and Indicators

The numeric data type can represent numbers of various types, such as integer or real. The two common numeric objects are the numeric control and the numeric indicator, as shown in Figure 1-8. Objects such as meters and dials also represent numeric data.

Figure 1-8. Numeric Controls and Indicators



- 1 Increment/Decrement Buttons
- 2 Numeric Control

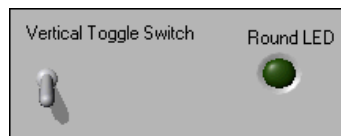
- 3 Numeric Indicator

To enter or change values in a numeric control, click the increment and decrement buttons with the Operating tool or double-click the number with either the Labeling tool or the Operating tool, enter a new number, and press the <Enter> key.

Boolean Controls and Indicators

The Boolean data type represents data that only has two possible states, such as TRUE and FALSE or ON and OFF. Use Boolean controls and indicators to enter and display Boolean values. Boolean objects simulate switches, push buttons, and LEDs. The vertical toggle switch and the round LED Boolean objects are shown in Figure 1-9.

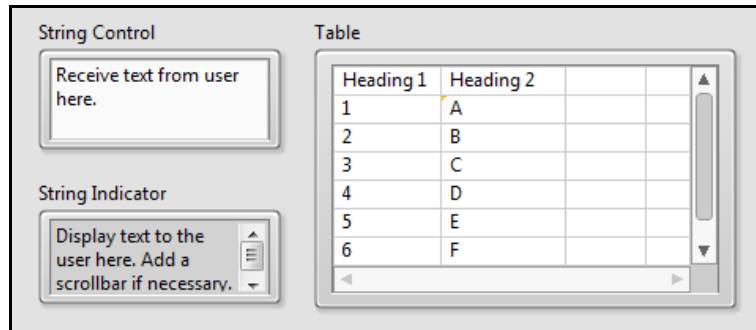
Figure 1-9. Boolean Controls and Indicators



String Controls and Indicators

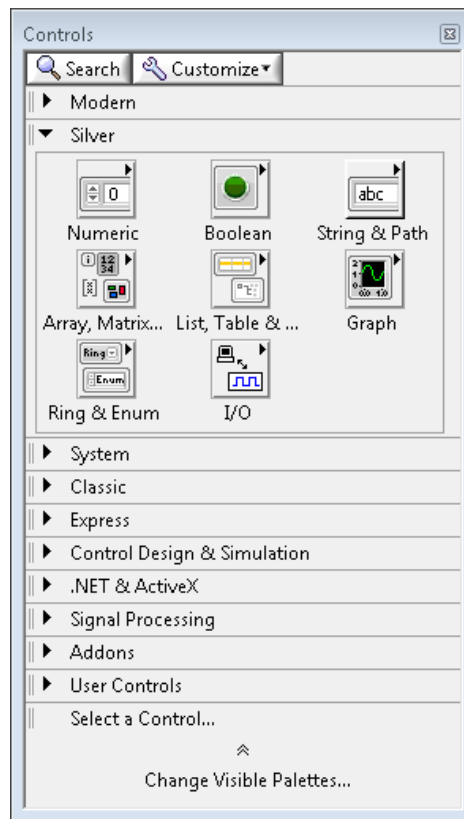
The string data type is a sequence of ASCII characters. Use string controls to receive text from the user such as a password or user name. Use string indicators to display text to the user. The most common string objects are tables and text entry boxes as shown in Figure 1-10.

Figure 1-10. String Controls and Indicators



Controls Palette

The **Controls** palette contains the controls and indicators you use to create the front panel. You access the **Controls** palette from the front panel window by selecting **View»Controls Palette**. The **Controls** palette is broken into various categories; you can expose some or all of these categories to suit your needs. Figure 1-11 shows a **Controls** palette with all of the categories exposed and the **Silver** category expanded. During this course, most exercises use controls from the **Silver** category.

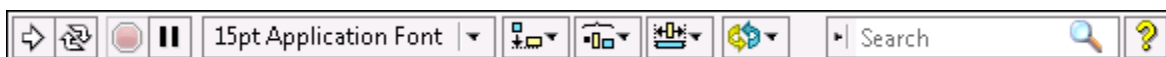
Figure 1-11. Controls Palette

To view or hide categories (subpalettes), select the **Customize** button on the palette, and select or deselect in the **Change Visible Palettes** option.

Front Panel Window Toolbar

Each window has a toolbar associated with it. Use the front panel window toolbar buttons to run and edit the VI.

The following toolbar appears on the front panel window.



Click the **Run** button to run a VI. LabVIEW compiles the VI, if necessary. You can run a VI if the **Run** button appears as a solid white arrow, shown below. The solid white arrow also indicates you can use the VI as a subVI if you create a connector pane for the VI.



While the VI runs, the **Run** button appears as shown below if the VI is a top-level VI, meaning it has no callers and therefore is not a subVI.



If the VI that is running is a subVI, the **Run** button appears as shown below.



The **Run** button appears broken when the VI you are creating or editing contains errors. If the **Run** button still appears broken after you finish wiring the block diagram, the VI is broken and cannot run. Click this button to display the **Error list** window, which lists all errors and warnings.



Click the **Run Continuously** button to run the VI until you abort or pause execution. You also can click the button again to disable continuous running.



While the VI runs, the **Abort Execution** button appears. Click this button to stop the VI immediately if there is no other way to stop the VI. If more than one running top-level VI uses the VI, the button is dimmed.



Caution The **Abort Execution** button stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design VIs with a stop button to avoid this problem.

Click the **Pause** button to pause a running VI. When you click the **Pause** button, LabVIEW highlights on the block diagram the location where you paused execution, and the **Pause** button appears red. Click the **Pause** button again to continue running the VI.



Select the **Text Settings** pull-down menu to change the font settings for the selected portions of the VI, including size, style, and color.



Select the **Align Objects** pull-down menu to align objects along axes, including vertical, top edge, left, and so on.



Select the **Distribute Objects** pull-down menu to space objects evenly, including gaps, compression, and so on.



Select the **Resize Objects** pull-down menu to resize multiple front panel objects to the same size.



Select the **Reorder** pull-down menu when you have objects that overlap each other and you want to define which one is in front or back of another. Select one of the objects with the Positioning tool and then select from **Move Forward**, **Move Backward**, **Move To Front**, and **Move To Back**.



Enter search terms to locate content in the *LabVIEW Help*.



Select the **Show Context Help Window** button to toggle the display of the **Context Help** window.



Enter Text appears to remind you that a new value is available to replace an old value. The **Enter Text** button disappears when you click it, press the <Enter> key, or click the front panel or block diagram workspace.

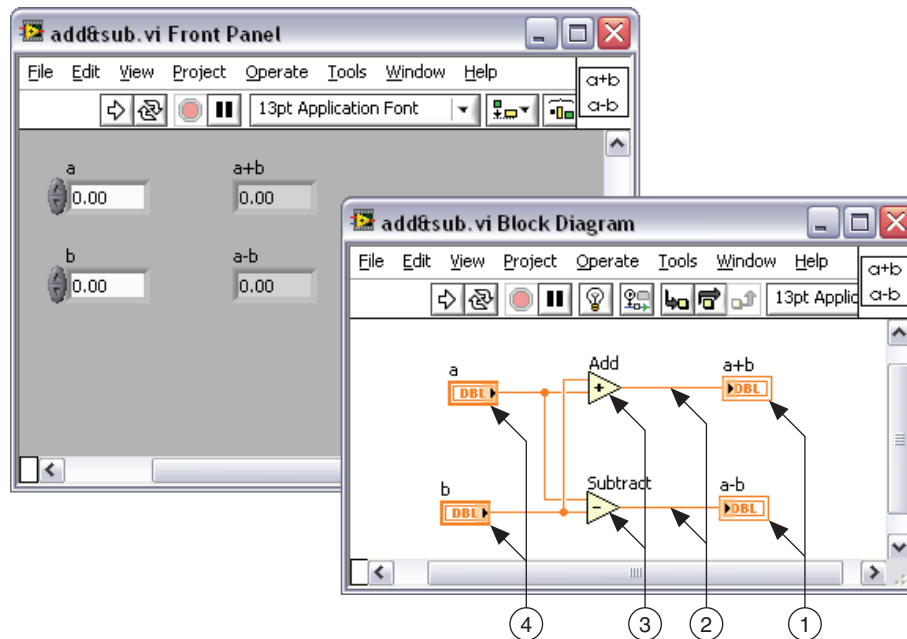


Tip The <Enter> key on the numeric keypad ends a text entry, while the main <Enter> key adds a new line. To modify this behavior, select **Tools»Options**, select the **Environment** from the **Category** list, and place a checkmark in the **End text entry with Enter key** option.

E. Block Diagram

Block diagram objects include terminals, subVIs, functions, constants, structures, and wires, which transfer data among other block diagram objects.

Figure 1-12. Example of a Block Diagram and Corresponding Front Panel



1 Indicator Terminals	2 Wires	3 Nodes	4 Control Terminals
-----------------------	---------	---------	---------------------

Terminals

Objects on the front panel window appear as terminals on the block diagram. Terminals are entry and exit ports that exchange information between the front panel and block diagram. Terminals are analogous to parameters and constants in text-based programming languages. Types of terminals include control or indicator terminals and node terminals. Control and indicator terminals belong to front panel controls and indicators. Data you enter into the front panel controls (**a** and **b** in the previous front panel) enter the block diagram through the control terminals. The data then enter the Add and Subtract functions. When the Add and Subtract functions complete their calculations, they produce new data values. The data values flow to the indicator terminals, where they update the front panel indicators (**a+b** and **a-b** in the previous front panel).

The terminals in Figure 1-12 belong to four front panel controls and indicators. Because terminals represent the inputs and outputs of your VI, subVIs and functions also have terminals shown below. For example, the connector panes of the Add and Subtract functions have three node terminals. To display the terminals of the function on the block diagram, right-click the function node and select **Visible Items»Terminals** from the shortcut menu.



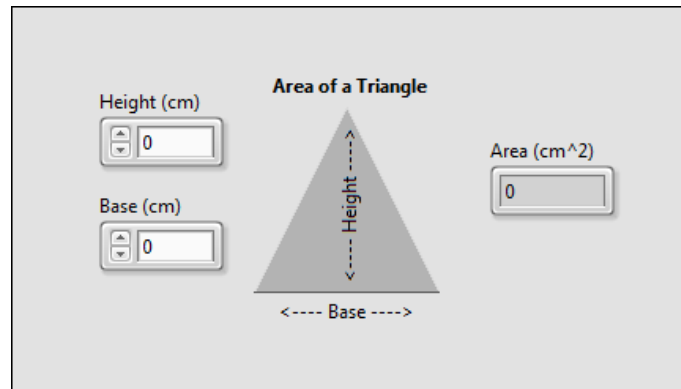
Controls, Indicators, and Constants

Controls, indicators, and constants behave as inputs and outputs of the block diagram algorithm. Consider the implementation of the algorithm for the area of a triangle:

$$\text{Area} = .5 * \text{Base} * \text{Height}$$

In this algorithm, **Base** and **Height** are inputs and **Area** is an output, as shown in Figure 1-13.

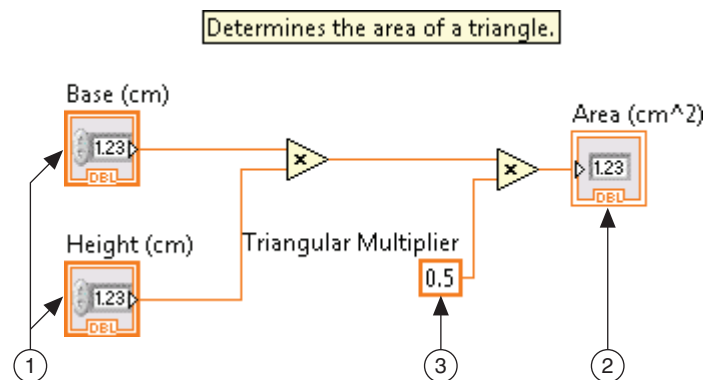
Figure 1-13. Area of a Triangle Front Panel



The user never changes or accesses the constant .5, so it does not appear on the front panel unless included as documentation of the algorithm.

Figure 1-14 shows a possible implementation of this algorithm on a LabVIEW block diagram. This block diagram has four different terminals created by two controls, one constant, and one indicator.

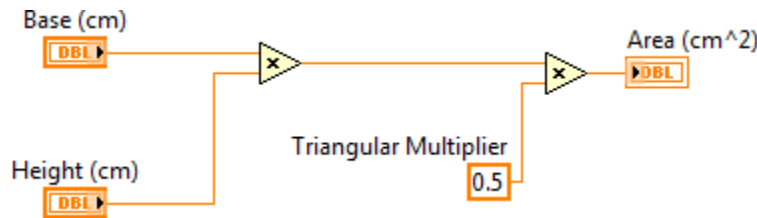
Figure 1-14. Area of a Triangle Block Diagram with Icon Terminal View



Notice that the **Base (cm)** and **Height (cm)** block diagram terminals have a different appearance from the **Area (cm²)** terminal. There are two distinguishing characteristics between a control and an indicator on the block diagram. The first is an arrow on the terminal that indicates the direction of data flow. The controls have arrows showing the data leaving the terminal, whereas the indicator has an arrow showing the data entering the terminal. The second distinguishing characteristic is the border around the terminal. Controls have a thick border and indicators have a thin border.

You can view terminals with or without icon view. Figure 1-15 shows the same block diagram without using the icon view of the terminals; however, the same distinguishing characteristics between controls and indicators exist.

Figure 1-15. Area of a Triangle Block Diagram without Icon Terminal View



Block Diagram Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. Nodes can be functions, subVIs, or structures. Structures are process control elements, such as Case structures, For Loops, or While Loops. The Add and Subtract functions in the Figure 1-12 are function nodes.

Functions

Functions are the fundamental operating elements of LabVIEW. Functions do not have front panel windows or block diagram windows but do have connector panes. Double-clicking a function only selects the function. A function has a pale yellow background on its icon.

SubVIs

SubVIs are VIs that you create to use inside of another VI or that you access on the **Functions** palette.

Any VI has the potential to be used as a subVI. When you double-click a subVI on the block diagram, its front panel window appears. The front panel includes controls and indicators. The block diagram includes wires, icons, functions, possibly subVIs, and other LabVIEW objects. The upper right corner of the front panel window and block diagram window displays the icon for the VI. This is the icon that appears when you place the VI on a block diagram as a subVI.

SubVIs can also be Express VIs. Express VIs are nodes that require minimal wiring because you configure them with dialog boxes. Use Express VIs for common measurement tasks. You can save the configuration of an Express VI as a subVI. Refer to the *Express VIs* topic of the *LabVIEW Help* for more information about creating a subVI from an Express VI configuration.

LabVIEW uses colored icons to distinguish between Express VIs and other VIs on the block diagram. Icons for Express VIs appear on the block diagram as icons surrounded by a blue field whereas subVI icons have a yellow field.

Expandable Nodes versus Icons

You can display VIs and Express VIs as icons or as expandable nodes. Expandable nodes appear as icons surrounded by a colored field. SubVIs appear with a yellow field, and Express VIs appear with a blue field. Use icons if you want to conserve space on the block diagram. Use expandable nodes to make wiring easier and to aid in documenting block diagrams. By default, subVIs appear as icons on the block diagram, and Express VIs appear as expandable nodes. To display a subVI or Express VI as an expandable node, right-click the subVI or Express VI and remove the checkmark next to the **View As Icon** shortcut menu item.

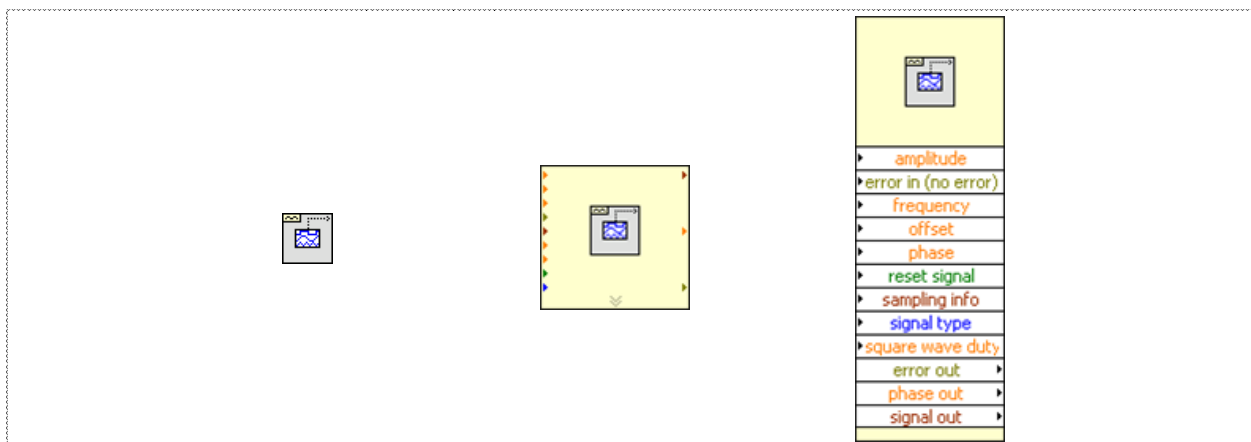
You can resize the expandable node to make wiring even easier, but it also takes a large amount of space on the block diagram. Complete the following steps to resize a node on the block diagram:

1. Move the Positioning tool over the node. Resizing handles appear at the top and bottom of the node.
2. Move the cursor over a resizing handle to change the cursor to the resizing cursor.
3. Use the resizing cursor to drag the border of the node down to display additional terminals.
4. Release the mouse button.

To cancel a resizing operation, drag the node border past the block diagram window before you release the mouse button.

Figure 1-16 shows the Basic Function Generator VI as a resized expandable node.

Figure 1-16. Basic Function Generator VI in Different Display Modes



Note If you display a subVI or Express VI as an expandable node, you cannot display the terminals for that node and you cannot enable database access for that node.

Wires

You transfer data among block diagram objects through wires. In Figure 1-12, wires connect the control and indicator terminals to the Add and Subtract function. Each wire has a single data source, but you can wire it to many VIs and functions that read the data. Wires are different colors, styles, and thicknesses, depending on their data types.

A broken wire appears as a dashed black line with a red X in the middle, as shown below. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types.



Table 1-1 shows the most common wire types.

Table 1-1. Common Wire Types

Wire Type	Scalar	1D Array	2D Array	Color
Numeric				Orange (floating-point), Blue (integer)
Boolean				Green
String				Pink

In LabVIEW, you use wires to connect multiple terminals together to pass data in a VI. You must connect the wires to inputs and outputs that are compatible with the data that is transferred with the wire. For example, you cannot wire an array output to a numeric input. In addition the direction of the wires must be correct. You must connect the wires to only one input and at least one output. For example, you cannot wire two indicators together. The components that determine wiring compatibility include the data type of the control and/or the indicator and the data type of the terminal.

Data Types

Data types indicate what objects, inputs, and outputs you can wire together. For example, if a switch has a green border, you can wire a switch to any input with a green label on an Express VI. If a knob has an orange border, you can wire a knob to any input with an orange label. However, you cannot wire an orange knob to an input with a green label. Notice the wires are the same color as the terminal.

Automatically Wiring Objects

As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to place the object on the block diagram, LabVIEW automatically connects the wires. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and does not connect the terminals that do not match.

Toggle automatic wiring by pressing the spacebar while you move an object using the Positioning tool.

By default, automatic wiring is enabled when you select an object from the **Functions** palette or when you copy an object already on the block diagram by pressing the <Ctrl> key and dragging the object. Automatic wiring is disabled by default when you use the Positioning tool to move an object already on the block diagram.

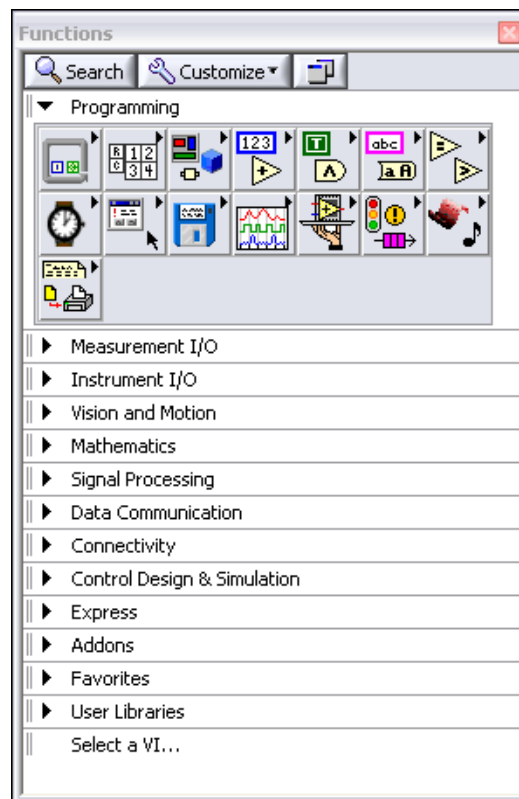
You can adjust the automatic wiring settings by selecting **Tools»Options** and selecting **Block Diagram** from the **Category** list.

Manually Wiring Objects

When you pass the Wiring tool over a terminal, a tip strip appears with the name of the terminal. In addition, the terminal blinks in the **Context Help** window and on the icon to help you verify that you are wiring to the correct terminal. To wire objects together, pass the Wiring tool over the first terminal, click, pass the cursor over the second terminal, and click again. After wiring, you can right-click the wire and select **Clean Up Wire** from the shortcut menu to have LabVIEW automatically choose a path for the wire. If you have broken wires to remove, press <Ctrl-B> to delete all the broken wires on the block diagram.

Functions Palette

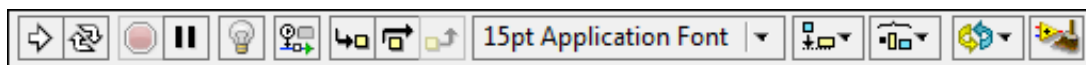
The **Functions** palette contains the VIs, functions and constants you use to create the block diagram. You access the **Functions** palette from the block diagram by selecting **View»Functions Palette**. The **Functions** palette is broken into various categories; you can show and hide categories to suit your needs. Figure 1-17 shows a **Functions** palette with all of the categories exposed and the **Programming** category expanded. During this course, you work mostly in the **Programming** category, but you also use other categories, or subpalettes.

Figure 1-17. Functions Palette

To view or hide categories, click the **Customize** button on the palette, and select or deselect the **Change Visible Palettes** option.

Block Diagram Toolbar

When you run a VI, buttons appear on the block diagram toolbar that you can use to debug the VI. The following toolbar appears on the block diagram.



Click the **Highlight Execution** button to display an animation of the block diagram execution when you run the VI. Notice the flow of data through the block diagram. Click the button again to disable execution highlighting.



Click the **Retain Wire Values** button to save the wire values at each point in the flow of execution so that when you place a probe on the wire you can immediately retain the most recent value of the data that passed through the wire. You must successfully run the VI at least once before you can retain the wire values.



Click the **Step Into** button to open a node and pause. When you click the **Step Into** button again, it executes the first action and pauses at the next action of the subVI or structure. You also can press the <Ctrl> and down arrow keys. Single-stepping through a VI steps through the VI node by node. Each node blinks to denote when it is ready to execute.



Click the **Step Over** button to execute a node and pause at the next node. You also can press the <Ctrl> and right arrow keys. By stepping over the node, you execute the node without single-stepping through the node.



Click the **Step Out** button to finish executing the current node and pause. When the VI finishes executing, the **Step Out** button is dimmed. You also can press the <Ctrl> and up arrow keys. By stepping out of a node, you complete single-stepping through the node and navigate to the next node.



Click the **Clean Up Diagram** button to automatically reroute all existing wires and rearrange objects on the block diagram to generate a cleaner layout. To configure the clean up options, select **Tools»Options** to display the Options dialog box and select **Block Diagram** from the **Category** list. You can configure the settings in the Block Diagram Cleanup section.



The **Warning** button appears if a VI includes a warning and you placed a checkmark in the **Show Warnings** checkbox in the **Error List** window. A warning indicates there is a potential problem with the block diagram, but it does not stop the VI from running.



LabVIEW Help Utilities

Use the **Context Help** window, the *LabVIEW Help*, and the NI Example Finder to help you create and edit VIs. Refer to the *LabVIEW Help* and manuals for more information about LabVIEW.

Context Help Window

The **Context Help** window displays basic information about LabVIEW objects when you move the cursor over each object. To toggle display of the **Context Help** window select **Help»Show Context Help**, press the <Ctrl-H> keys, or click the **Show Context Help Window** button on the toolbar.

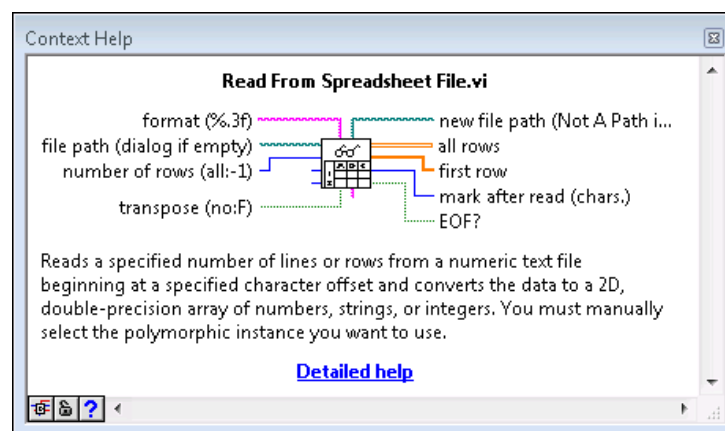


When you move the cursor over front panel and block diagram objects, the **Context Help** window displays the icon for subVIs, functions, constants, controls, and indicators, with wires attached to each terminal. When you move the cursor over dialog box options, the **Context Help** window displays descriptions of those options.

In the **Context Help** window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button in the **Context Help** window.

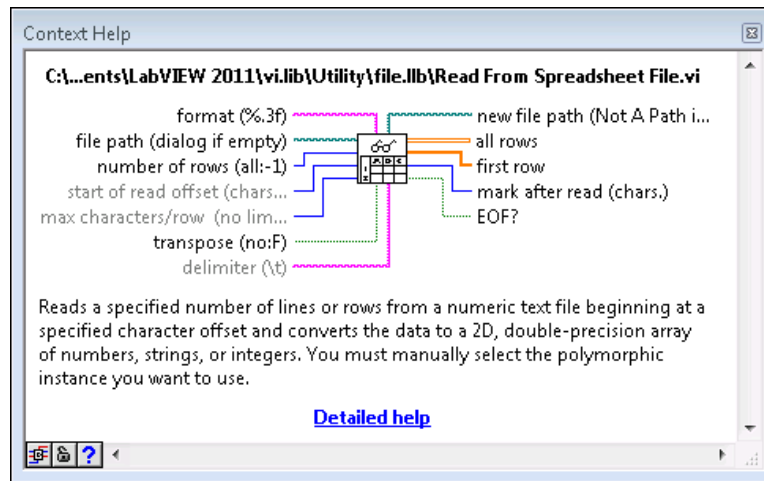


Figure 1-18. Context Help Window



Click the **Show Optional Terminals and Full Path** button located on the lower left corner of the **Context Help** window to display the optional terminals of a connector pane and to display the full path to a VI. Optional terminals are shown by wire stubs, informing you that other connections exist. The detailed mode displays all terminals, as shown in Figure 1-19.



Figure 1-19. Detailed Context Help Window

Click the **Lock Context Help** button to lock the current contents of the **Context Help** window. When the contents are locked, moving the cursor over another object does not change the contents of the window. To unlock the window, click the button again. You also can access this option from the **Help** menu.



If a corresponding *LabVIEW Help* topic exists for an object the **Context Help** window describes, a blue **Detailed help** link appears in the **Context Help** window. Also, the **More Help** button is enabled. Click the link or the button to display the *LabVIEW Help* for more information about the object.



LabVIEW Help

You can access the *LabVIEW Help* by clicking the **More Help** button in the **Context Help** window, selecting **Help»LabVIEW Help**, or clicking the blue **Detailed Help** link in the **Context Help** window. You also can right-click an object and select **Help** from the shortcut menu.

The *LabVIEW Help* contains detailed descriptions of most palettes, menus, tools, VIs, and functions. The *LabVIEW Help* also includes step-by-step instructions for using LabVIEW features. The *LabVIEW Help* includes links to the following resources:

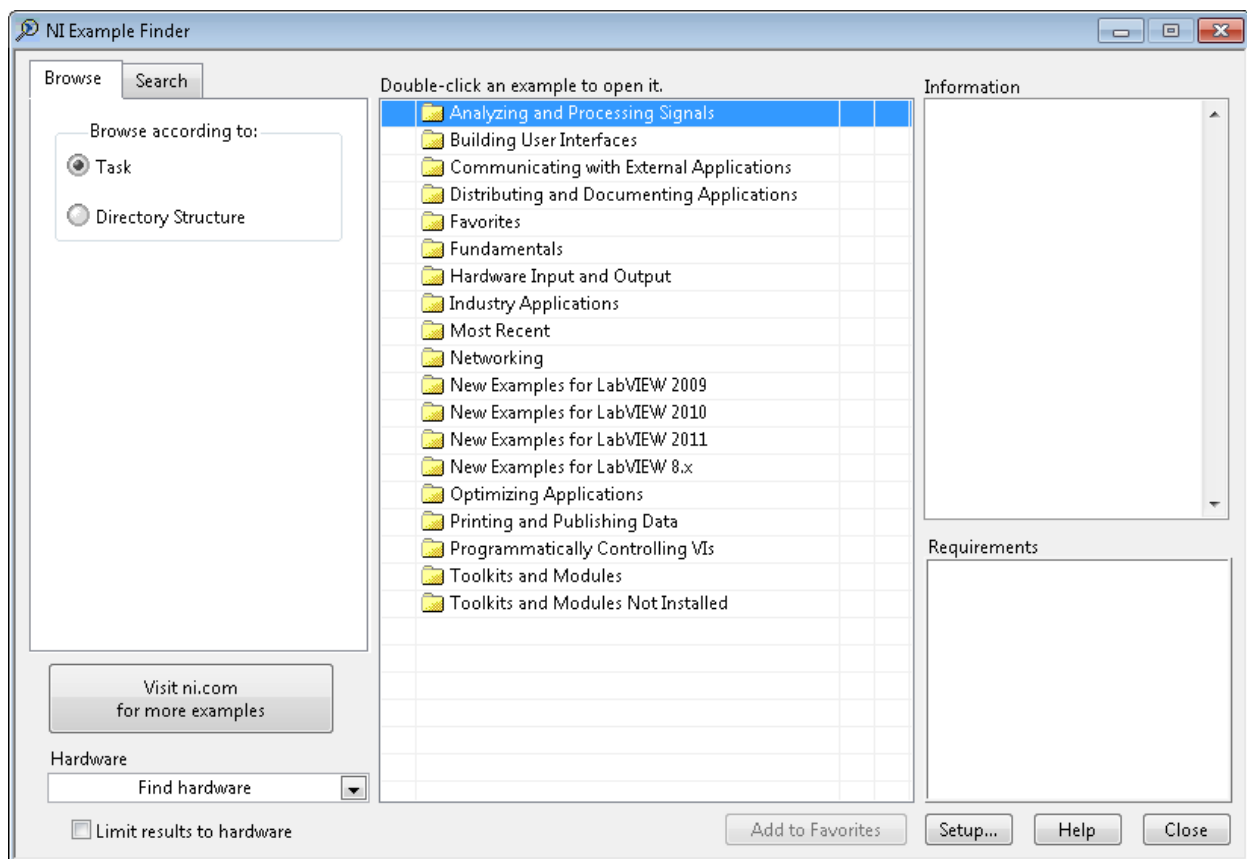
- *LabVIEW Documentation Resources*, which describes online and print documents to help new and experienced users and includes PDF versions of all LabVIEW manuals.
- Technical support resources on the National Instruments Web site, such as the NI Developer Zone, the KnowledgeBase, and the Product Manuals Library.

NI Example Finder

Use the NI Example Finder to browse or search examples installed on your computer or on the NI Developer Zone at ni.com/zone. These examples demonstrate how to use LabVIEW to perform a wide variety of test, measurement, control, and design tasks. Select **Help»Find Examples** or click the **Find Examples** link in the **Examples** section of the **Getting Started** window to launch the NI Example Finder.

Examples can show you how to use specific VIs or functions. You can right-click a VI or function on the block diagram or on a pinned palette and select **Examples** from the shortcut menu to display a help topic with links to examples for that VI or function. You can modify an example VI to fit an application, or you can copy and paste from one or more examples into a VI that you create.

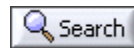
Figure 1-20. NI Example Finder



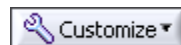
F. Searching for Controls, VIs and Functions

When you select **View»Controls** or **View»Functions** to open the **Controls** and **Functions** palettes, two buttons appear at the top of the palette.

Search—Changes the palette to search mode so you can perform text-based searches to locate controls, VIs, or functions on the palettes. While a palette is in search mode, click the Return button to exit search mode and return to the palette.

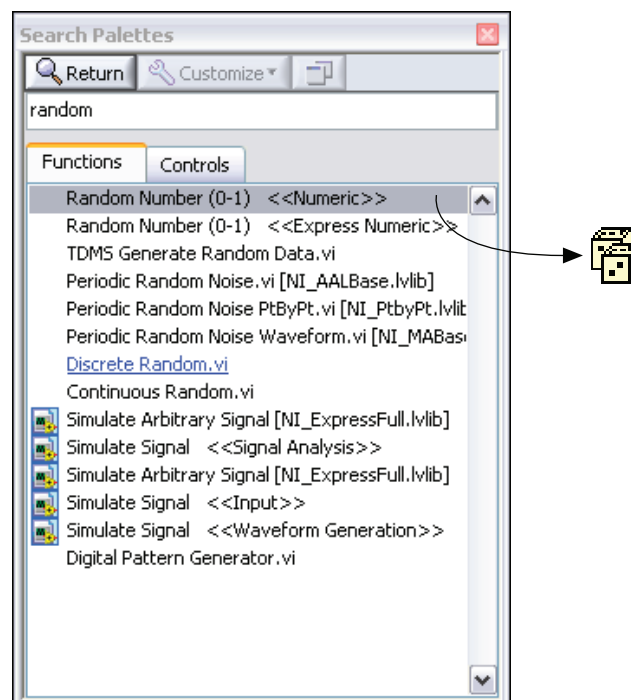


Customize—Provides options for selecting a format for the current palette, showing and hiding categories for all palettes, and sorting items in the **Text** and **Tree** formats alphabetically. Select **Options** from the shortcut menu to display the **Controls/Functions Palettes** page of the **Options** dialog box, in which you can select a format for all palettes. This button appears only if you click the thumbtack in the upper left corner of a palette to pin the palette.



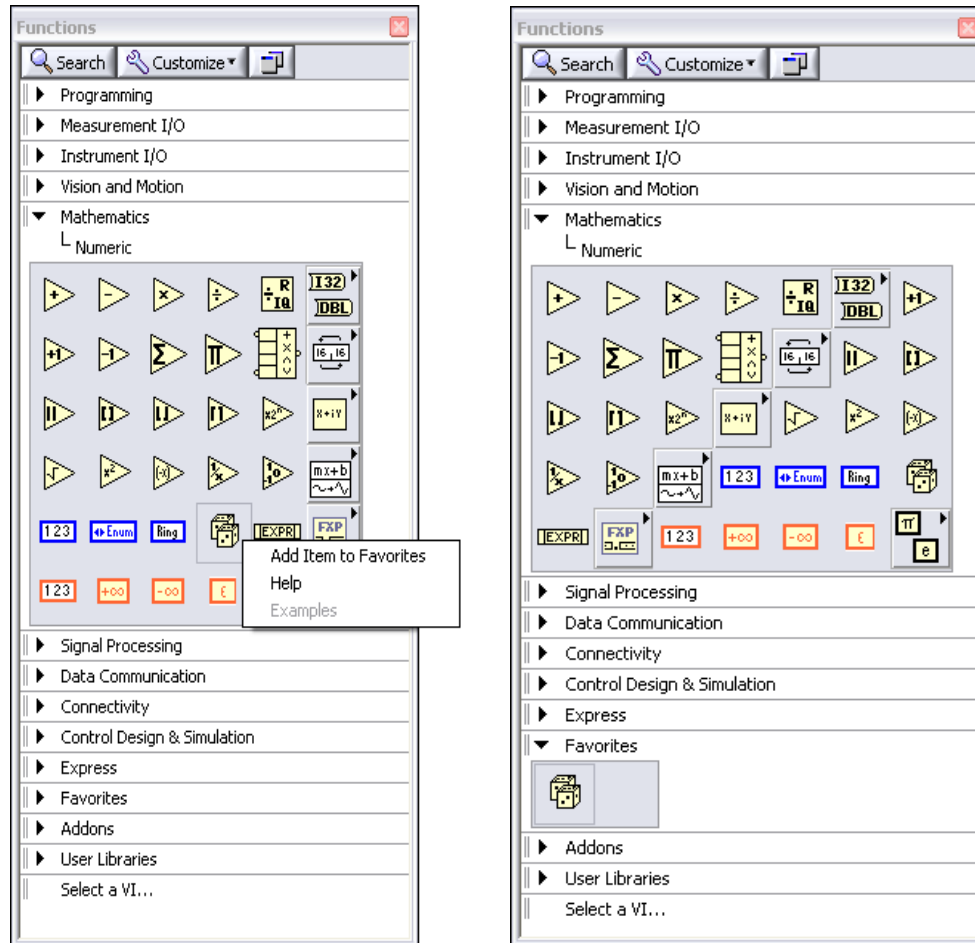
Until you are familiar with the location of VIs and functions, search for the function or VI using the **Search** button. For example, if you want to find the Random Number function, click the **Search** button on the **Functions** palette toolbar and start typing Random Number in the text box at the top of the palette. LabVIEW lists all matching items that either start with or contain the text you typed. You can click one of the search results and drag it to the block diagram, as shown in Figure 1-21.

Figure 1-21. Searching for an Object in the Functions Palette



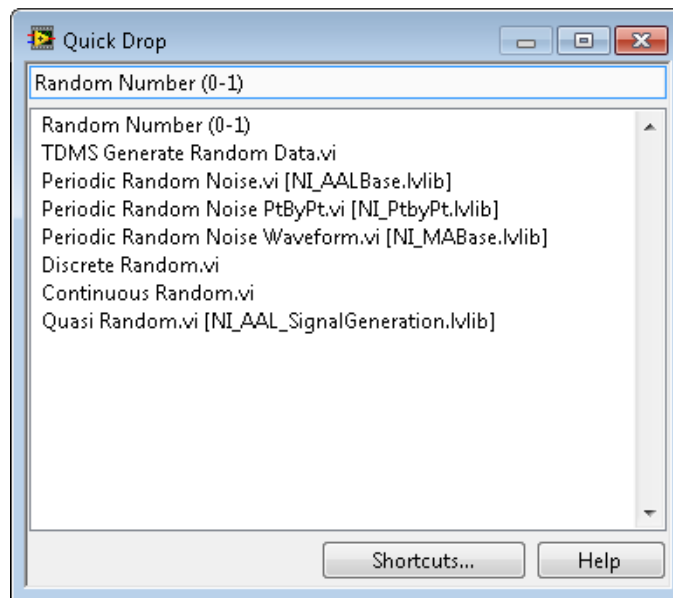
Double-click the search result to highlight its location on the palette. If the object is one you need to use frequently, you can add it to your Favorites category. Right-click the object on the palette and select **Add Item to Favorites**, as shown in Figure 1-22.

Figure 1-22. Adding an Item to the Favorites Category of a Palette



Similar to the Search button, you use the **Quick Drop** dialog box to specify a palette object by name and then place the object on the block diagram or front panel. In addition to palette objects, you can also specify a project item by name in the **Quick Drop** dialog box.

To display the **Quick Drop** dialog box, press <Ctrl-Space> or select **View»Quick Drop**. Type the name of the object you want to add to the block diagram or front panel. LabVIEW displays the results in the **Name Match List**. To attach the object you select to the cursor, press the <Enter> key, double-click the name of the object in the list, or click the block diagram or front panel. Click the location on the block diagram or front panel where you want to add the object.

Figure 1-23. Searching for an Object in the Quick Drop Dialog Box

G. Selecting a Tool

You can create, modify and debug VIs using the tools provided by LabVIEW. A tool is a special operating mode of the mouse cursor. The operating mode of the cursor corresponds to the icon of the tool selected. LabVIEW chooses which tool to select based on the current location of the mouse.

Figure 1-24. Tools Palette

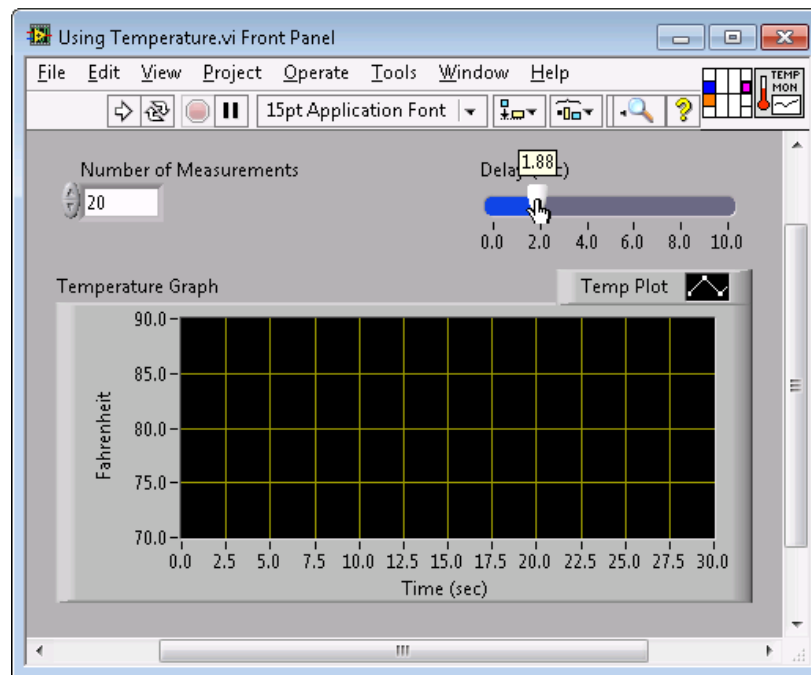
Tip You can manually choose the tool you need by selecting it on the **Tools** palette. Select **View»Tools Palette** to display the **Tools** palette.

Operating Tool

When the mouse cursor changes to the icon shown below, the Operating tool is in operation. Use the Operating tool to change the values of a control. For example, in Figure 1-25 the Operating tool moves the pointer on the Horizontal Pointer Slide. When the mouse hovers over the pointer, the cursor automatically accesses the Operating tool.



Figure 1-25. Using the Operating Tool

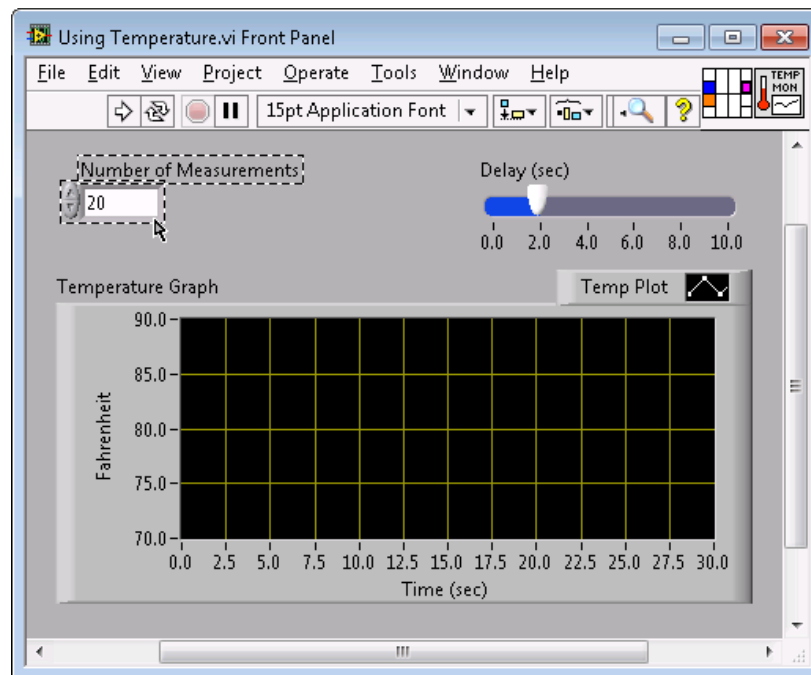


The Operating tool is mostly used on the front panel window, but you also can use the Operating tool on the block diagram window to change the value of a Boolean constant.

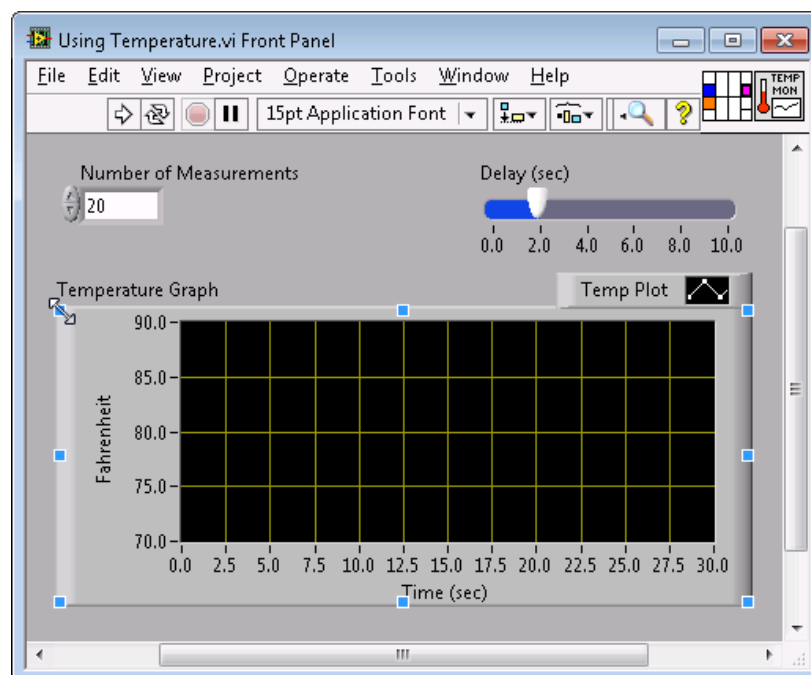
Positioning Tool

When the mouse cursor changes to the icon shown below, the Positioning tool is in operation. Use the Positioning tool to select or resize objects. For example, in Figure 1-26 the Positioning tool selects the **Number of Measurements** numeric control. After selecting an object, you can move, copy, or delete the object. When the mouse hovers over the edge of an object, the cursor automatically accesses the Positioning tool.



Figure 1-26. Using the Positioning Tool to Select an Object

If the mouse hovers over a resizing node of an object, the cursor mode changes to show that you can resize the object, as shown in Figure 1-27. Notice that the cursor is hovering over a corner of the XY Graph at a resizing node, and the cursor mode changes to a double-sided arrow.

Figure 1-27. Using the Positioning Tool to Resize an Object

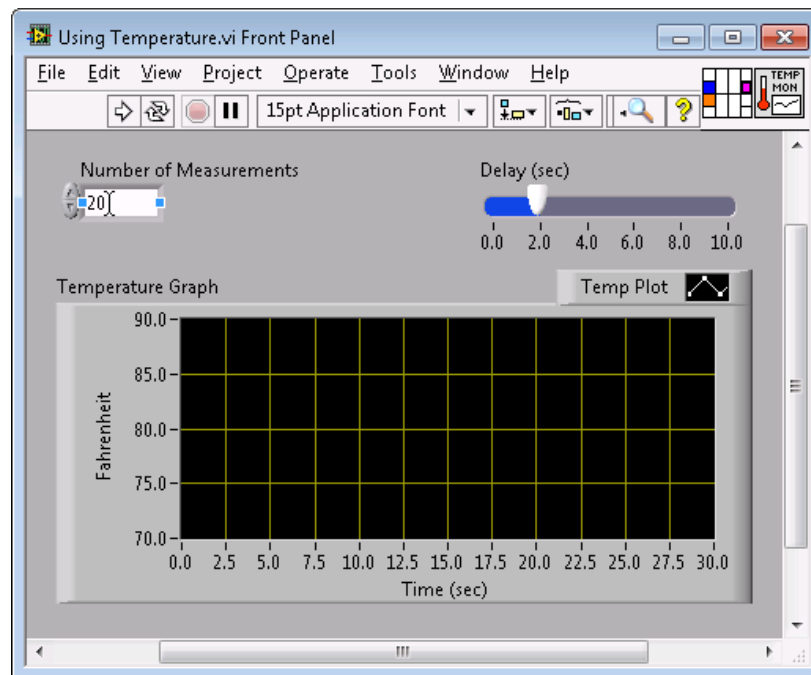
You can use the Positioning tool on both the front panel window and the block diagram.

Labeling Tool

When the mouse cursor changes to the icon shown below, the Labeling tool is in operation. Use the Labeling tool to enter text in a control, to edit text, and to create free labels. For example, in Figure 1-28 the Labeling tool enters text in the **Number of Measurements** numeric control. When the mouse hovers over the interior of the control, the cursor automatically accesses the Labeling tool. Click once to place a cursor inside the control. Then double-click to select the current text.



Figure 1-28. Using the Labeling Tool



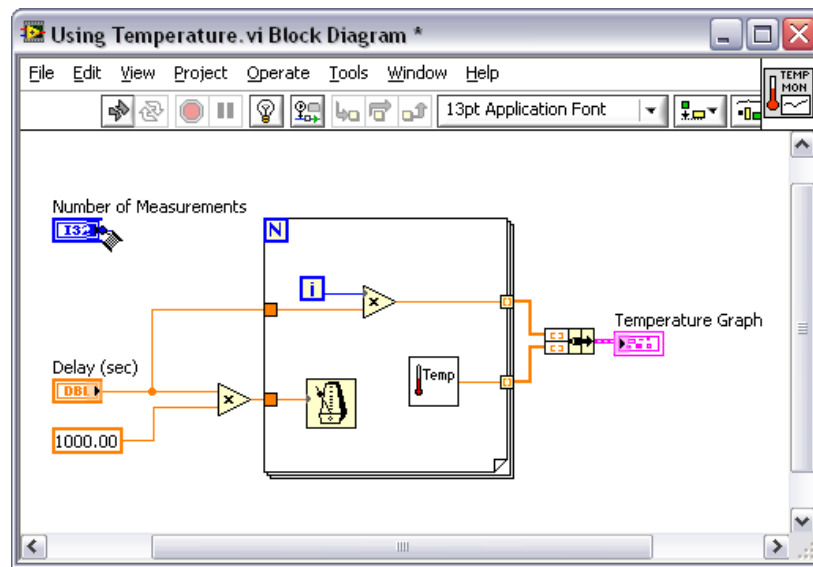
When you are not in a specific area of a front panel window or block diagram window that accesses a certain mouse mode, the cursor appears as cross-hairs. If automatic tool selection is enabled, you can double-click any open space to access the Labeling tool and create a free label.

Wiring Tool

When the mouse cursor changes to the icon shown below, the Wiring tool is in operation. Use the Wiring tool to wire objects together on the block diagram. For example, in Figure 1-29 the Wiring tool wires the **Number of Measurements** terminal to the count terminal of the For Loop. When the mouse hovers over the exit or entry point of a terminal or over a wire, the cursor automatically accesses the Wiring tool.



Figure 1-29. Using the Wiring Tool



The Wiring tool works mainly with the block diagram window and when you create a connector pane on the front panel window.

Other Tools Accessed from the Palette

You can access the Operating, Positioning, Labeling, and Wiring tools directly from the **Tools** palette, rather than using the Automatic tool selection mode. Select **View»Tools Palette** to access the **Tools** palette.

Figure 1-30. The Tools Palette



The top item in the **Tools** palette is the **Automatic Tool Selection** button. When this is selected, LabVIEW automatically chooses a tool based on the location of your cursor. You can turn off automatic tool selection by deselecting the item, or by selecting another item in the palette. There are some additional tools on the palette, as described below:



Use the Object Shortcut Menu tool to access an object shortcut menu with the left mouse button.



Use the Scrolling tool to scroll through windows without using scrollbars.



Use the Breakpoint tool to set breakpoints on VIs, functions, nodes, wires, and structures to pause execution at that location.



Use the Probe tool to create probes on wires on the block diagram. Use the Probe tool to check intermediate values in a VI that produces questionable or unexpected results.



Use the Color Copy tool to copy colors for pasting with the Coloring tool.



Use the Coloring tool and the color picker to color an object. The Coloring tool also displays the current foreground and background color settings.



Select the Coloring tool and right-click an object or workspace to display the color picker.

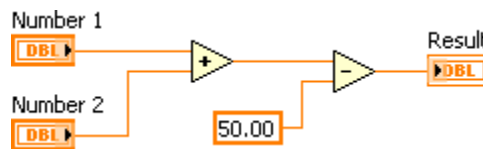
H. Dataflow

LabVIEW follows a dataflow model for running VIs. A block diagram node executes when it receives all required inputs. When a node executes, it produces output data and passes the data to the next node in the dataflow path. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.

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

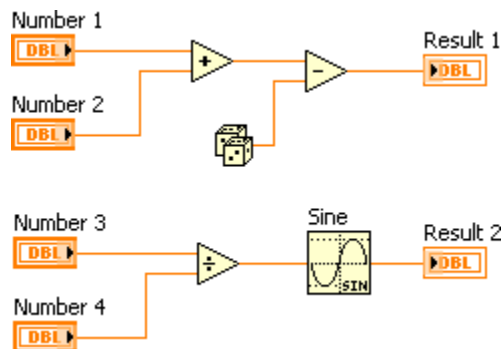
For a dataflow programming example, consider a block diagram that adds two numbers and then subtracts 50.00 from the result of the addition, as shown in Figure 1-31. In this case, the block diagram executes from left to right, not because the objects are placed in that order, but because the Subtract function cannot execute until the Add function finishes executing and passes the data to the Subtract function. Remember that a node executes only when data are available at all of its input terminals and supplies data to the output terminals only when the node finishes execution.

Figure 1-31. Dataflow Programming Example



In Figure 1-32, consider which code segment would execute first—the Add, Random Number, or Divide function. You cannot know because inputs to the Add and Divide functions are available at the same time, and the Random Number function has no inputs. In a situation where one code segment must execute before another, and no data dependency exists between the functions, use other programming methods, such as error clusters, to force the order of execution. Refer to Lesson 5, *Creating and Leveraging Data Structures*, for more information about error clusters.

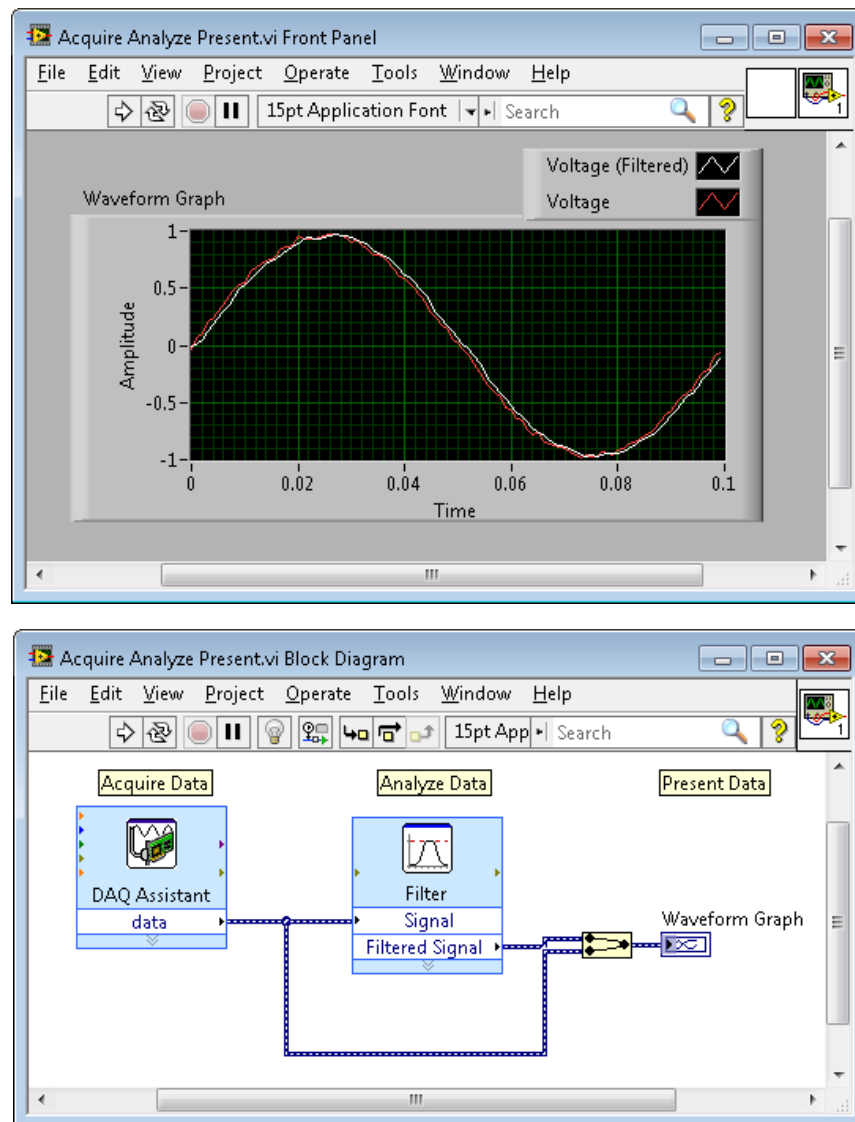
Figure 1-32. Dataflow Example for Multiple Code Segments



I. Building a Simple VI

Most LabVIEW VIs have three main tasks—acquiring some sort of data, analyzing the acquired data, and presenting the result. When each of these parts are simple, you can complete the entire VI using very few objects on the block diagram. Express VIs are designed specifically for completing common, frequently used operations. In this section, you learn about some Express VIs that acquire, analyze, and present data. Then you learn to create a simple VI that uses these three tasks, as shown in Figure 1-33.

Figure 1-33. Acquire, Analyze, and Present Example Front Panel Window and Block Diagram Window



On the **Functions** palette, the Express VIs are grouped together in the **Express** category. Express VIs use the dynamic data type to pass data between Express VIs.

Acquire

Express VIs used for the Acquire Data task include the following: DAQ Assistant, Instrument I/O Assistant, Simulate Signal, and Read from Measurement File.

DAQ Assistant

The DAQ Assistant acquires data through a data acquisition device. You must use this Express VI frequently throughout this course. Until you learn more about data acquisition, you only use one channel of the data acquisition device, CH0. This channel is connected to a temperature sensor on the BNC-2120. You can touch the temperature sensor to change the temperature the sensor reads.



Instrument I/O Assistant

The Instrument I/O Assistant acquires instrument control data, usually from a GPIB or serial interface.



Simulate Signal

The Simulate Signal Express VI generates simulated data such as a sine wave.



Read From Measurement File

The Read From Measurement File Express VI reads a file that was created using the Write To Measurement File Express VI. It specifically reads LVM or TDM file formats. This Express VI does not read ASCII files. Refer to Lesson 6, *Managing File and Hardware Resources*, for more information on reading data from a file.



Analyze

Express VIs used for the Analyze Data task include the following—Amplitude and Level Measurements, Statistics, and Tone Measurements.

Amplitude and Level Measurements

The Amplitude and Level Measurements Express VI performs voltage measurements on a signal. These include DC, rms, maximum peak, minimum peak, peak to peak, cycle average, and cycle rms measurements.



Statistics

The Statistics Express VI calculates statistical data from a waveform. This includes mean, sum, standard deviation, and extreme values.



Spectral Measurements

The Spectral Measurements Express VI performs spectral measurement on a waveform, such as magnitude and power spectral density.



Tone Measurements

The Tone Measurements Express VI searches for a single tone with the highest frequency or highest amplitude. It also finds the frequency and amplitude of a single tone.



Filter

The Filter Express VI processes a signal through filters and windows. Filters used include the following: Highpass, Lowpass, Bandpass, Bandstop, and Smoothing. Windows used include Butterworth, Chebyshev, Inverse Chebyshev, Elliptical, and Bessel.



Present

Present results by using Express VIs that perform a function, such as the Write to Measurement File Express VI, or indicators that present data on the front panel window. The most commonly used indicators for this task include the Waveform Chart, the Waveform Graph, and the XY Graph. Common Express VIs include the Write to Measurement File Express VI, the Build Text Express VI, the DAQ Assistant, and the Instrument I/O Assistant. In this case, the DAQ Assistant and the Instrument I/O Assistant provide output data from the computer to the DAQ device or an external instrument.

Write to Measurement File

The Write to Measurement File Express VI writes a file in LVM or TDMS file format. Refer to Lesson 6, *Managing File and Hardware Resources*, for more information on writing to measurement files.



Build Text

The Build Text Express VI creates text, usually for displaying on the front panel window or exporting to a file or instrument. Refer to Lesson 6, *Managing File and Hardware Resources*, for more information on creating strings.



Running a VI

After you configure the Express VIs and wire them together, you can run the VI. When you finish creating your VI, click the **Run** button on the toolbar to execute the VI.



While the VI is running, the **Run** button icon changes to the figure shown below. After the execution completes, the **Run** button icon changes back to its original state, and the front panel indicators contain data.



Run Button Errors

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button appears broken when the VI you are creating or editing contains errors.



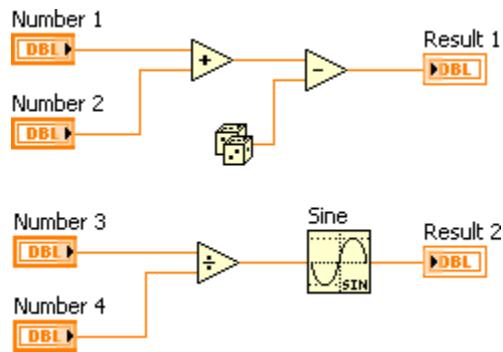
If the button still appears broken when you finish wiring the block diagram, the VI is broken and cannot run.

Generally, this means that a required input is not wired, or a wire is broken. Press the broken run button to access the **Error list** window. The **Error list** window lists each error and describes the problem. You can double-click an error to go directly to the error. Refer to Lesson 2, *Troubleshooting and Debugging VIs*, for more information on debugging VIs.

Self-Review: Quiz

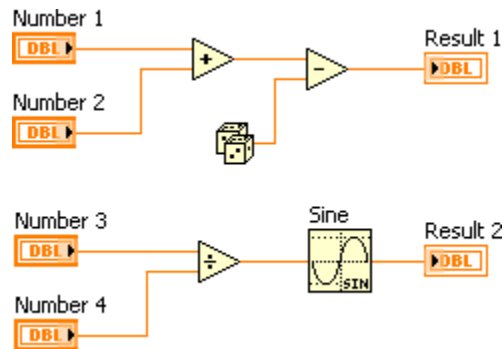
Refer to Figure 1-34 to answer the following quiz questions.

Figure 1-34. Dataflow Questions



1. Which function executes first: Add or Subtract?
 - a. Add
 - b. Subtract
 - c. Unknown
2. Which function executes first: Sine or Divide?
 - a. Sine
 - b. Divide
 - c. Unknown
3. Which function executes first: Random Number, Divide or Add?
 - a. Random Number
 - b. Divide
 - c. Add
 - d. Unknown
4. Which function executes last: Random Number, Subtract or Add?
 - a. Random Number
 - b. Subtract
 - c. Add
 - d. Unknown
5. What are the three parts of a VI?
 - a. Front panel window
 - b. Block diagram window
 - c. Project
 - d. Icon/connector pane

Self-Review: Quiz Answers



1. Which function executes first: Add or Subtract?
 - a. Add**
 - b. Subtract
 - c. Unknown
2. Which function executes first: Sine or Divide?
 - a. Sine
 - b. Divide**
 - c. Unknown
3. Which function executes first?
 - a. Random Number
 - b. Divide
 - c. Add
 - d. Unknown**
4. Which function executes last: Random, Subtract or Add?
 - a. Random Number
 - b. Subtract**
 - c. Add
 - d. Unknown
5. What are the three parts of a VI?
 - a. Front panel window**
 - b. Block diagram window**
 - c. Project
 - d. Icon/connector pane**

Notes

Troubleshooting and Debugging VIs

To run a VI, you must wire all the subVIs, functions, and structures with the correct data types for the terminals. Sometimes a VI produces data or runs in a way you do not expect. You can use LabVIEW to configure how a VI runs and to identify problems with block diagram organization or with the data passing through the block diagram.

Topics

- A. Correcting Broken VIs
- B. Debugging Techniques
- C. Undefined or Unexpected Data
- D. Error Handling

A. Correcting Broken VIs

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button appears broken when the VI you are creating or editing contains errors.



If the button still appears broken when you finish wiring the block diagram, the VI is broken and cannot run.

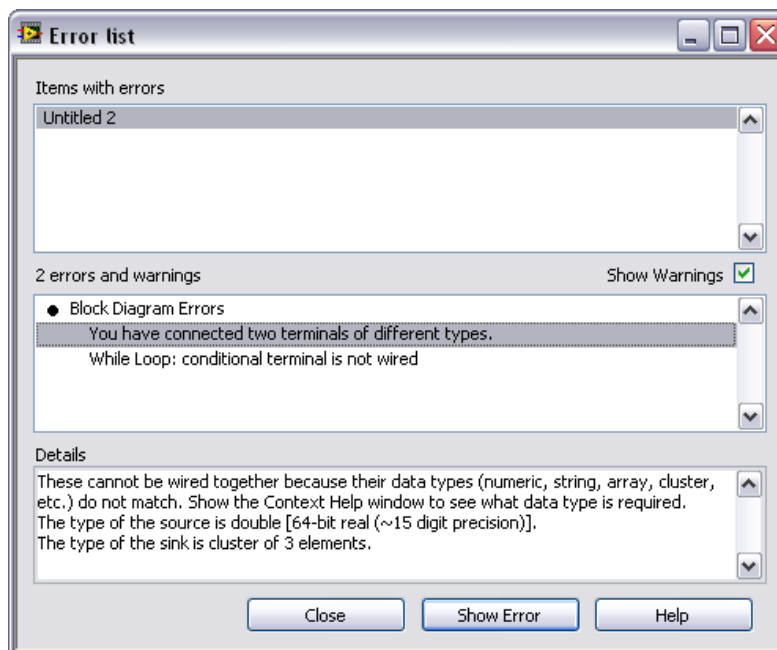
Finding Causes for Broken VIs

Warnings do not prevent you from running a VI. They are designed to help you avoid potential problems in VIs. Errors, however, can break a VI. You must resolve any errors before you can run the VI.

Click the broken **Run** button or select **View»Error List** to find out why a VI is broken. The **Error list** window lists all the errors. The **Items with errors** section lists the names of all items in memory, such as VIs and project libraries that have errors. If two or more items have the same name, this section shows the specific application instance for each item. The **errors and warnings** section lists the errors and warnings for the VI you select in the **Items with errors** section. The **Details** section describes the errors and in some cases recommends how to correct the errors. Click the **Help** button to display a topic in the *LabVIEW Help* that describes the error in detail and includes step-by-step instructions for correcting the error.

Click the **Show Error** button or double-click the error description to highlight the area on the block diagram or front panel that contains the error.

Figure 2-1. Example of the Error List Dialog Box



Common Causes of Broken VIs

The following list contains common reasons why a VI is broken while you edit it:

- The block diagram contains a broken wire because of a mismatch of data types or a loose, unconnected end. Refer to the *Correcting Broken VIs* topic of the *LabVIEW Help* for information about correcting broken wires.
- A required block diagram terminal is unwired. Refer to the *Using Wires to Link Block Diagram Objects* topic of the *LabVIEW Help* for information about setting required inputs and outputs.
- A subVI is broken or you edited its connector pane after you placed its icon on the block diagram of the VI. Refer to the *Creating SubVIs* topic of the *LabVIEW Help* for information about subVIs.

B. Debugging Techniques

If a VI is not broken, but you get unexpected data, you can use the following techniques to identify and correct problems with the VI or the block diagram data flow:

- Wire the error in and error out parameters at the bottom of most built-in VIs and functions. These parameters detect errors encountered in each node on the block diagram and indicate if and where an error occurred. You also can use these parameters in the VIs you build.
- To eliminate all VI warnings, select **View»Error List** and place a checkmark in the **Show Warnings** checkbox to see all warnings for the VI. Determine the causes and correct them in the VI.
- Use the Positioning tool to triple-click a wire to highlight its entire path and to ensure that the wires connect to the proper terminals.
- Use the **Context Help** window to check the default values for each function and subVI on the block diagram. VIs and functions pass default values if recommended or optional inputs are unwired. For example, a Boolean input might be set to TRUE if unwired.
- Use the **Find** dialog box to search for subVIs, text, and other objects to correct throughout the VI.
- Select **View»VI Hierarchy** to find unwired subVIs. Unlike unwired functions, unwired VIs do not always generate errors unless you configure an input to be required. If you mistakenly place an unwired subVI on the block diagram, it executes when the block diagram does. Consequently, the VI might perform extra actions.
- Use execution highlighting to watch the data move through the block diagram.
- Single-step through the VI to view each action of the VI on the block diagram.
- Use the Probe tool to observe intermediate data values and to check the error output of VIs and functions, especially those performing I/O.
- Click the **Retain Wire Values** button on the block diagram toolbar to retain wire values for use with probes. This feature allows you to easily check values of data that last passed through any wire.
- Use breakpoints to pause execution, so you can single-step or insert probes.

- Suspend the execution of a subVI to edit values of controls and indicators, to control the number of times it runs, or to go back to the beginning of the execution of the subVI.
- Determine if the data that one function or subVI passes is undefined. This often happens with numbers. For example, at one point in the VI an operation could have divided a number by zero, thus returning **Inf** (infinity), whereas subsequent functions or subVIs were expecting numbers.
- If the VI runs more slowly than expected, confirm that you turned off execution highlighting in subVIs. Also, close subVI front panels and block diagrams when you are not using them because open windows can affect execution speed.
- Check the representation of controls and indicators to see if you are receiving overflow because you converted a floating-point number to an integer or an integer to a smaller integer. For example, you might wire a 16-bit integer to a function that only accepts 8-bit integers. This causes the function to convert the 16-bit integer to an 8-bit representation, potentially causing a loss of data.
- Determine if any For Loops inadvertently execute zero iterations and produce empty arrays.
- Verify you initialized shift registers properly unless you intend them to save data from one execution of the loop to another.
- Check the cluster element order at the source and destination points. LabVIEW detects data type and cluster size mismatches at edit time, but it does not detect mismatches of elements of the same type.
- Check the node execution order.
- Check that the VI does not contain hidden subVIs. You inadvertently might have hidden a subVI by placing one directly on top of another node or by decreasing the size of a structure without keeping the subVI in view.
- Check the inventory of subVIs the VI uses against the results of **View»Browse Relationships»This VI's SubVIs** and **View»Browse Relationships»Unopened SubVIs** to determine if any extra subVIs exist. Also open the VI Hierarchy window to see the subVIs for a VI. To help avoid incorrect results caused by hidden VIs, specify that inputs to VIs are required.

Execution Highlighting

View an animation of the execution of the block diagram by clicking the **Highlight Execution** button.



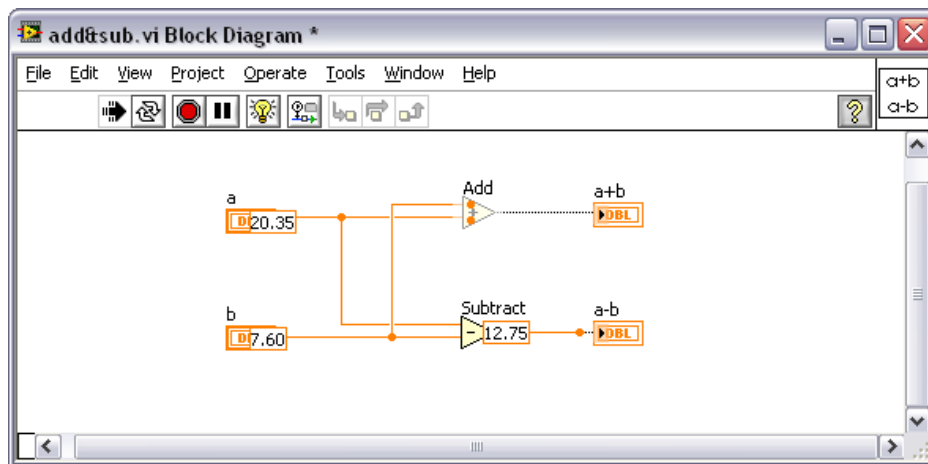
Execution highlighting shows the movement of data on the block diagram from one node to another using bubbles that move along the wires. Use execution highlighting in conjunction with single-stepping to see how data values move from node to node through a VI.

(MathScript RT Module) In MathScript Nodes, execution highlighting shows the progression from one line of script to another using a blue arrow that blinks next to the line that is executing currently.



Note Execution highlighting greatly reduces the speed at which the VI runs.

Figure 2-2. Example of Execution Highlighting in Use



Single-Stepping

Single-step through a VI to view each action of the VI on the block diagram as the VI runs. The single-stepping buttons, shown as follows, affect execution only in a VI or subVI in single-step mode.



Enter single-step mode by clicking the **Step Into** or **Step Over** button on the block diagram toolbar. Move the cursor over the **Step Into**, **Step Over**, or **Step Out** button to view a tip strip that describes the next step if you click that button. You can single-step through subVIs or run them normally.

When you single-step through a VI, nodes blink to indicate they are ready to execute. If you single-step through a VI with execution highlighting on, an execution glyph appears on the icons of the subVIs that are currently running.



Probe Tools

Use the Probe tool to check intermediate values on a wire as a VI runs.



Use the Probe tool if you have a complicated block diagram with a series of operations, any one of which might return incorrect data. Use the Probe tool with execution highlighting, single-stepping, and breakpoints to determine if and where data is incorrect. If data is available, the probe immediately updates and displays the data in the **Probe Watch Window** during execution highlighting, single-stepping, or when you pause at a breakpoint. When execution pauses at a node because of single-stepping or a breakpoint, you also can probe the wire that just executed to see the value that flowed through that wire.



Tip If you want a probe to display the data that flowed through the wire during the last VI execution, click the **Retain Wire Values** button on the block diagram toolbar.

Types of Probes

You can check intermediate values on a wire when a VI runs by using a generic probe, by using an indicator on the **Controls** palette to view the data, by using a supplied probe, by using a customized supplied probe, or by creating a new probe.



Note (MathScript RT Module) You can view the data in a script in a MathScript Node as a VI runs by using a LabVIEW MathScript probe.

Generic

Use the generic probe to view the data that passes through a wire. Right-click a wire and select **Custom Probe»Generic Probe** from the shortcut menu to use the generic probe.

The generic probe displays the data. You cannot configure the generic probe to respond to the data.

LabVIEW displays the generic probe when you right-click a wire and select **Probe**, unless you already specified a custom or supplied probe for the data type.

You can debug a custom probe similar to a VI. However, a probe cannot probe its own block diagram, nor the block diagram of any of its subVIs. When debugging probes, use the generic probe.

Using Indicators to View Data

You also can use an indicator to view the data that passes through a wire. For example, if you view numeric data, you can use a chart within the probe to view the data. Right-click a wire, select **Custom Probe»Controls** from the shortcut menu, and select the indicator you want to use. You also can click the **Select a Control** icon on the **Controls** palette and select any custom control or

type definition saved on the computer or in a shared directory on a server. LabVIEW treats type definitions as custom controls when you use them to view probed data.

If the data type of the indicator you select does not match the data type of the wire you right-clicked, LabVIEW does not place the indicator on the wire.

Supplied

Supplied probes are VIs that display comprehensive information about the data that passes through a wire. For example, the VI Refnum Probe returns information about the VI name, the VI path, and the hex value of the reference. You also can use a supplied probe to respond based on the data that flows through the wire. For example, use an Error probe on an error cluster to receive the status, code, source, and description of the error and specify if you want to set a conditional breakpoint if an error or warning occurs.

The supplied probes appear at the top of the **Custom Probe** shortcut menu. Right-click a wire and select **Custom Probe** from the shortcut menu to select a supplied probe. Only probes that match the data type of the wire you right-click appear on the shortcut menu.

Refer to the Using Supplied Probes VI in the `labview\examples\general\probes.llb` for an example of using supplied probes.

Custom

Use the **Create New Probe** dialog box to create a probe based on an existing probe or to create a new probe. Right-click a wire and select **Custom Probe»New** from the shortcut menu to display the **Create New Probe** dialog box. Create a probe when you want to have more control over how LabVIEW probes the data that flows through a wire. When you create a new probe, the data type of the probe matches the data type of the wire you right-clicked. If you want to edit the probe you created, you must open it from the directory where you saved it.

After you select a probe from the **Custom Probe** shortcut menu, navigate to it using the **Select a Control** palette option, or create a new probe using the **Create New Probe** dialog box, that probe becomes the default probe for that data type, and LabVIEW loads that probe when you right-click a wire and select **Probe** from the shortcut menu. LabVIEW only loads probes that exactly match the data type of the wire you right-click. That is, a double precision floating-point numeric probe cannot probe a 32-bit unsigned integer wire even though LabVIEW can convert the data.



Note If you want a custom probe to be the default probe for a particular data type, save the probe in the `user.lib_probes\default` directory. Do not save probes in the `vi.lib_probes` directory because LabVIEW overwrites those files when you upgrade or reinstall.

Breakpoints

Use the Breakpoint tool to place a breakpoint on a VI, node, or wire and pause execution at that location.



When you set a breakpoint on a wire, execution pauses after data passes through the wire and the **Pause** button appears red. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute. The block diagram border appears red and blinks to reflect the placement of a breakpoint.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and uses a marquee to highlight the node, wire, or line of script that contains the breakpoint. When you move the cursor over an existing breakpoint, the black area of the Breakpoint tool cursor appears white.

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

- Single-step through execution using the single-stepping buttons.
- Probe wires to check intermediate values.
- Change the values of front panel controls.
- Click the **Pause** button to continue running to the next breakpoint or until the VI finishes running.

Suspending Execution

Suspend execution of a subVI to edit values of controls and indicators, to control the number of times the subVI runs before returning to the caller, or to go back to the beginning of the execution of the subVI. You can cause all calls to a subVI to start with execution suspended, or you can suspend a specific call to a subVI.

To suspend all calls to a subVI, open the subVI and select **Operate»Suspend when Called**. The subVI automatically suspends when another VI calls it. If you select this menu item when single-stepping, the subVI does not suspend immediately. The subVI suspends when it is called.

To suspend a specific subVI call, right-click the subVI node on the block diagram and select **SubVI Node Setup** from the shortcut menu. Place a checkmark in the **Suspend when called** checkbox to suspend execution only at that instance of the subVI.

The **VI Hierarchy** window, which you display by selecting **View»VI Hierarchy**, indicates whether a VI is paused or suspended.

An arrow glyph indicates a VI that is running regularly or single-stepping.



A pause glyph indicates a paused or suspended VI.



- A green pause glyph, or a hollow glyph in black and white, indicates a VI that pauses when called.
- A red pause glyph, or a solid glyph in black and white, indicates a VI that is currently paused.

An exclamation point glyph indicates that the subVI is suspended.



Note A VI can be suspended and paused at the same time.

Determining the Current Instance of a SubVI

When you pause a subVI, the **Call list** pull-down menu on the toolbar lists the chain of callers from the top-level VI down to the subVI. This list is not the same list you see when you select **View» Browse Relationships» This VI's Callers**, which lists all calling VIs regardless of whether they are currently running. Use the **Call list** menu to determine the current instance of the subVI if the block diagram contains more than one instance. When you select a VI from the **Call list** menu, its block diagram opens and LabVIEW highlights the current instance of the subVI.

You also can use the Call Chain function to view the chain of callers from the current VI to the top-level VI.

C. Undefined or Unexpected Data

Undefined data, which are `NaN` (not a number) or `Inf` (infinity), invalidate all subsequent operations. Floating-point operations return the following two symbolic values that indicate faulty computations or meaningless results:

- `NaN` (not a number) represents a floating-point value that invalid operations produce, such as taking the square root of a negative number.
- `Inf` (infinity) represents a floating-point value that valid operations produce, such as dividing a number by zero.

LabVIEW does not check for overflow or underflow conditions on integer values. Overflow and underflow for floating-point numbers is in accordance with IEEE 754, *Standard for Binary Floating-Point Arithmetic*.

Floating-point operations propagate `NaN` and `Inf` reliably. When you explicitly or implicitly convert `NaN` or `Inf` to integers or Boolean values, the values become meaningless. For example, dividing 1 by zero produces `Inf`. Converting `Inf` to a 16-bit integer produces the value 32,767, which appears to be a normal value.

Before you convert data to integer data types, use the Probe tool to check intermediate floating-point values for validity. Check for NaN by wiring the Not A Number/Path/Refnum? function to the value you suspect is invalid.

Do not rely on special values such as NaN, Inf, or empty arrays to determine if a VI produces undefined data. Instead, confirm that the VI produces defined data by making the VI report an error if it encounters a situation that is likely to produce undefined data.

For example, if you create a VI that uses an incoming array to auto-index a For Loop, determine what you want the VI to do when the input array is empty. Either produce an output error code, substitute defined data for the value that the loop creates, or use a Case structure that does not execute the For Loop if the array is empty.

D. Error Handling

No matter how confident you are in the VI you create, you cannot predict every problem a user can encounter. Without a mechanism to check for errors, you know only that the VI does not work properly. Error checking tells you why and where errors occur.

Error handling is the mechanism for anticipation, detection, and resolution of warnings and errors. Error handling is an essential component in your LabVIEW application development. With error handling you quickly pinpoint the source of programming errors. Without it, you might observe unexpected behavior but struggle to find the source of the problem.

Error handling is also extremely valuable when you test your application to ensure that your error reporting is meaningful and that the error handling code safely stops your application when an error occurs. For example, during stress testing you are setting values or conditions that are beyond the normal operational capacity of your application which often result in errors. When such errors occur, you want to ensure proper shutdown of your application.

Error handling continues to be important after an application is deployed. Error handling can help detect system and environment differences—such as differences in file systems, memory, and disk resources. National Instruments strongly recommends using error handling.

Automatic Error Handling

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box. In the error dialog box, each error has a numeric code to identify it and a corresponding error message to display to the user.

To disable automatic error handling for the current VI, select **File»VI Properties** and select **Execution** from the **Category** pull-down menu. To disable automatic error handling for any new, blank VIs you create, select **Tools»Options** and select **Block Diagram** from the **Category** list. To disable automatic error handling for a subVI or function within a VI, wire its **error out** parameter to the **error in** parameter of another subVI or function or to an **error out** indicator.

Manual Error Handling

You may choose to handle errors without relying on automatic error handling. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop and display an error dialog box. You also might want the VI to retry for a certain period of time.

In LabVIEW, you can make these manual error handling decisions on the block diagram of the VI using the following methods:

- Use the LabVIEW error handling VIs and functions on the **Dialog & User Interface** palette to notify and prompt users. For example, if LabVIEW encounters an error, you can display the error message in different kinds of dialog boxes.
- Use error clusters and the **error in** and **error out** parameters of most VIs and functions to manage errors. For example, when an error is detected you can fix the error programmatically and then wire the **error out** output of the VI or function to the **error in** input of the Clear Errors VI.



Tip Use error handling in conjunction with the debugging tools to find and manage errors.

Error Clusters

VIs and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs. Use the error cluster controls and indicators to create error inputs and outputs in subVIs.



Tip All error clusters on a block diagram typically provide the same standard error in and standard error out functionality.

The standard **error in** and **error out** clusters include the following components of information:

- **status** is a Boolean value that reports TRUE if an error occurred.
- **code** is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a **status** of FALSE signals a warning rather than an error.
- **source** is a string that identifies where the error occurred.

Errors

An error is defined as an error cluster with a **status** value of TRUE, regardless of the code value.

If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code.

Warnings

A warning is defined as an error cluster with a nonzero **code** value and a FALSE **status** value. Although most errors have negative code values and warnings have positive code values, this is not universally true. Therefore you should rely on both the **status** value and the **code** value to detect errors and warnings.

Warnings are typically considered less severe than errors. Some APIs and functions, such as the Match Regular Expression function, only report errors. However, other APIs, such as the VISA API for controlling stand-alone instruments, often reports warnings.

Unlike when an error occurs, nodes execute normally when LabVIEW detects a warning. Even though code executes normally, it is important that you monitor warnings during development to ensure proper behavior of your application.

Explain Error Dialog Box

When an error occurs, right-click within the cluster border and select **Explain Error** from the shortcut menu to open the **Explain Error** dialog box. The **Explain Error** dialog box contains information about the error. The shortcut menu includes an **Explain Warning** option if the VI contains warnings but no errors.

You also can access the **Explain Error** dialog box from the **Help»Explain Error** menu.

Detect and Report Errors

Error handling in LabVIEW follows the dataflow model. Just as data values flow through a VI, so can error information.

To implement good error handling, you must determine the actions to take when an error occurs at any point in your application. To begin with, you must utilize the error terminals on functions and VIs. Since the error cluster is implemented as a flow-through parameter, you should propagate errors by wiring the **error out** cluster of the first node you want to execute to the **error in** cluster of the next node you want to execute. You must continue to do this for sequences of nodes.

As the VI runs, LabVIEW tests for errors at each node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error or warning, the node passes the error to the next node. Any subVIs that you create should also implement this flow-through behavior.

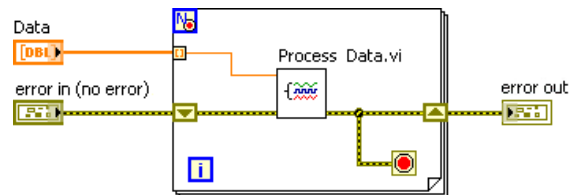


Tip When you perform any kind of input and output (I/O), consider the possibility that errors might occur. Almost all I/O functions return error information. Include error checking in VIs, especially for I/O operations (file, serial, instrumentation, data acquisition, and communication), and provide a mechanism to handle errors appropriately.

Propagating Errors and Warnings

To ensure that error and warning information is propagated correctly, it is important to use shift-registers for error cluster wires in loops so that the warning information is propagated through all iterations. Refer to Figure 2-3 for proper use of the shift-register to propagate errors and warnings to successive loop iterations.

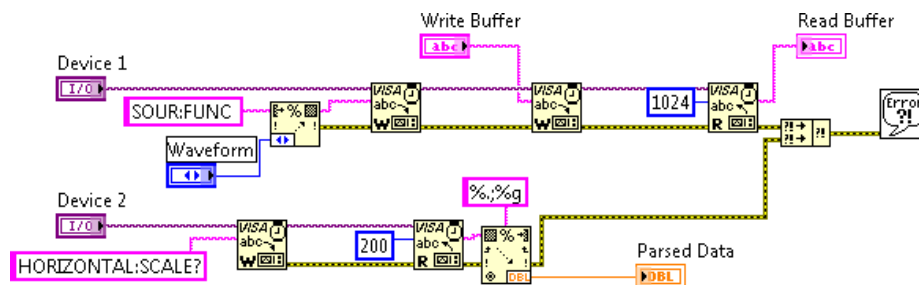
Figure 2-3. Use Shift Registers to Propagate Errors and Warnings



Merging Errors and Warnings

Use the Merge Error function to merge the **error out** cluster values from parallel sequences. Refer to Figure 2-4 for an example of merging error information from parallel node sequences.

Figure 2-4. Merge Errors From Multiple Sources



At the end of your application after all error sources are merged into one error cluster, you must report errors to the user using the Simple Error Handler VI or another error reporting mechanism.

Simple Error Handler VI

By default, the Simple Error Handler VI displays a dialog with a description of any errors that occurred and does not report warnings. However, the Simple Error Handler VI can be configured for other error handling behavior. You can select the type of dialog from the following options:

- **No dialog**—Displays no dialog box. This is useful if you want to have programmatic control over handling errors.
- **OK message (default)**—Displays a dialog box with a single OK button. After the user acknowledges the dialog box, the VI returns control to the main VI.
- **Continue or stop message**—Displays a dialog box with buttons, which the user can use to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.

- **OK message with warnings**—Displays a dialog box with any warnings and a single OK button. After the user acknowledges the dialog box, the VI returns control to the main VI.
- **Continue/Stop + Warnings**—Displays a dialog box with any warnings and buttons, which the user can use to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.

Ranges of Error Codes

VIs and functions in LabVIEW can return numeric error codes. Each product or group of VIs defines a range of error codes. Refer to the *Ranges of LabVIEW Error Codes* topic of the *LabVIEW Help* for error code tables listing the numeric error codes and descriptions for each product and VI grouping.

In addition to defining error code ranges, LabVIEW reserves some error code ranges for you to use in your application. You can define custom error codes in the range of –8999 through –8000, 5000 through 9999, or 500,000 through 599,999.

Some numeric error codes are used by more than one group of VIs and functions. For example, error 65 is both a serial error code, indicating a serial port timeout, and a networking error code, indicating that a network connection is already established.

Self-Review: Quiz

1. Which of the following will result in a broken run arrow?
 - a. A subVI is broken
 - b. The diagram includes a divide by zero
 - c. A required subVI input is unwired
 - d. A Boolean terminal is wired to a numeric indicator

2. Which of the following are the components and data types of the error cluster?
 - a. Status: Boolean
 - b. Error: String
 - c. Code: 32-bit integer
 - d. Source: String

3. All errors have negative error codes and all warnings have positive error codes.
 - a. True
 - b. False

4. Merge Errors function concatenates error information from multiple sources.
 - a. True
 - b. False

Self-Review: Quiz Answers

1. Which of the following will result in a broken run arrow?
 - a. **A subVI is broken**
 - b. The diagram includes a divide by zero
 - c. **A required subVI input is unwired**
 - d. **A Boolean terminal is wired to a numeric indicator**

2. Which of the following are the contents of the error cluster?
 - a. **Status: Boolean**
 - b. Error: String
 - c. **Code: 32-bit integer**
 - d. **Source: String**

3. All errors have negative error codes and all warnings have positive error codes.
 - a. True
 - b. **False**

4. Merge Errors function concatenates error information from multiple sources.
 - a. True
 - b. **False**

Notes

Implementing a VI

This lesson teaches you how to implement code in LabVIEW. These skills include designing a user interface, choosing a data type, documenting your code, using looping structures such as While Loops and For Loops, adding software timing to your code, displaying your data as a plot, and making decisions in your code using a Case structure.

Topics

- A. Front Panel Basics
- B. LabVIEW Data Types
- C. Documenting Code
- D. While Loops
- E. For Loops
- F. Timing a VI
- G. Data Feedback in Loops
- H. Plotting Data – Waveform Chart
- I. Case Structures

A. Front Panel Basics

In the design phase of the software development method, you identify the inputs and outputs of the problem. This identification leads directly to the design of the front panel window.

Inputs for the design can come from the following actions:

- acquiring from a device such as a data acquisition device or a multimeter
- reading directly from a file
- manipulating controls on the front panel

You can display the inputs of the problem on the front panel with controls, such as numeric, boolean, or string controls, but not all input appears on the front panel.

You can display the outputs of the problem with indicators, such as graphs, charts, or LEDs, or log the outputs to a file. You also can output data to a device using signal generation.

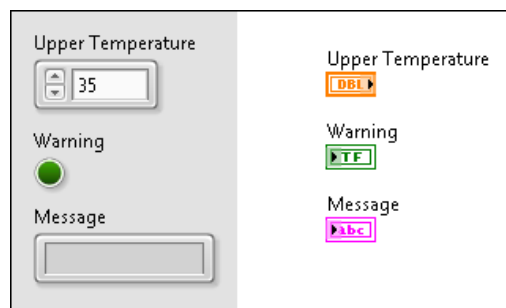
Designing Controls and Indicators

When choosing controls and indicators, make sure that they are appropriate for the task you want to perform. For example, when you want to determine the frequency of a sine wave, choose a dial control, or when you want to display temperature, choose a thermometer indicator.

Labels

Make sure to label controls and indicators clearly. These labels help users identify the purpose of each control and indicator. Also, clear labeling helps you document your code on the block diagram. Control and indicator labels correspond to the names of terminals on the block diagram, as shown in Figure 3-1.

Figure 3-1. Front Panel Controls and Indicators Appear on Block Diagram



1 Front Panel Window

2 Block Diagram

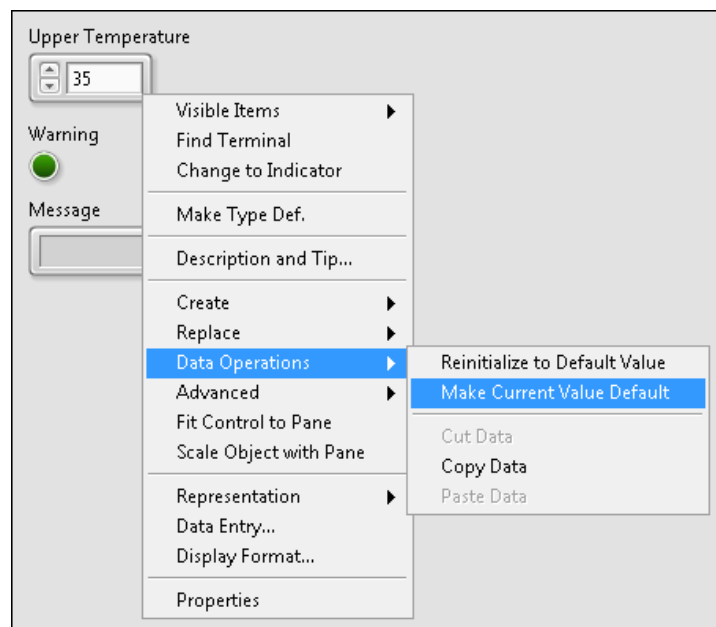
Control and Indicator Options

You can set default values for controls and indicators. Figure 3-2 shows the **Upper Temperature** control with a default value of 35 °C. By setting a default value, you can assume a reasonable value for a VI if the user does not set another value during run-time.

Complete the following steps to set the default value of a control or indicator:

1. Enter the desired value.
2. Right-click the control or indicator and select **Data Operations»Make Current Value Default** from the shortcut menu.

Figure 3-2. Setting Default Values



Tip You can also reinitialize values to their default values.



Tip To initialize or re-initialize all controls and indicators on the front panel at the same time, select **Edit»Make Current Values Default** or **Edit»Reinitialize to Default Values** from the LabVIEW menu.

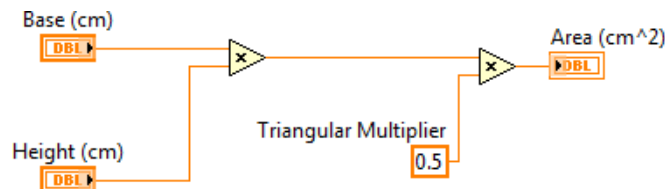
B. LabVIEW Data Types

Many different data types exist for data. You already learned about numeric, Boolean, and string data types in Lesson 1, *Navigating LabVIEW*. Other data types include the enumerated data type, dynamic data, and others. Even within numeric data types, there are different data types, such as whole numbers or fractional numbers.

Block Diagram Terminals and Data Types

The block diagram terminals visually communicate to the user some information about the data type the terminals represent. For example, in Figure 3-3, **Height (cm)** is a double-precision, floating-point numeric. This is indicated by the color of the terminal, orange, and by the text shown on the terminal, **DBL**.

Figure 3-3. Terminal Data Type Example



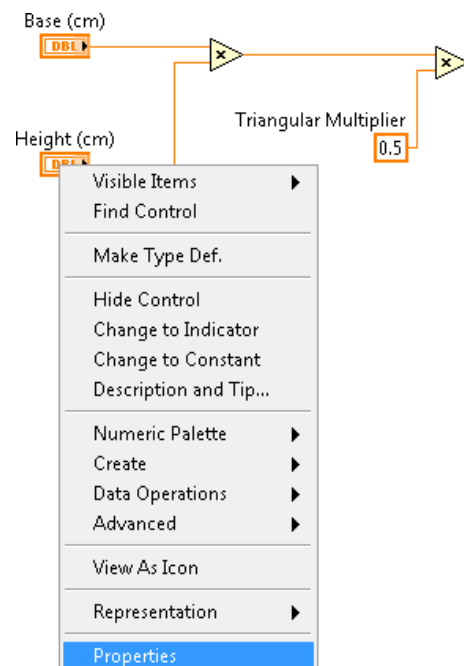
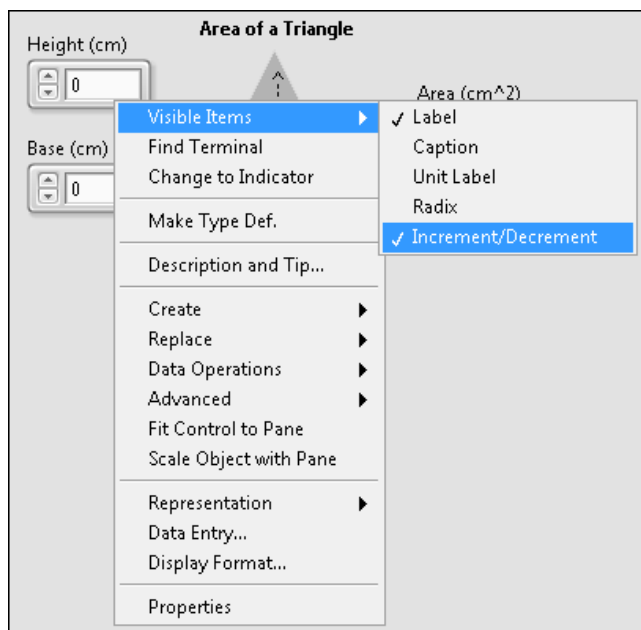
Tip Terminal names correspond to the labels of the controls and indicators on the front panel. Right-click a terminal and select **Find control** or **Find indicator** from the shortcut menu to locate the control or indicator on the front panel.

Shortcut Menus

All LabVIEW objects have associated shortcut menus, also known as context menus, pop-up menus, and right-click menus. As you create a VI, use the shortcut menu items to change the appearance or behavior of front panel and block diagram objects. To access the shortcut menu, right-click the object.

Figure 3-4 shows a shortcut menu for a control and terminal.

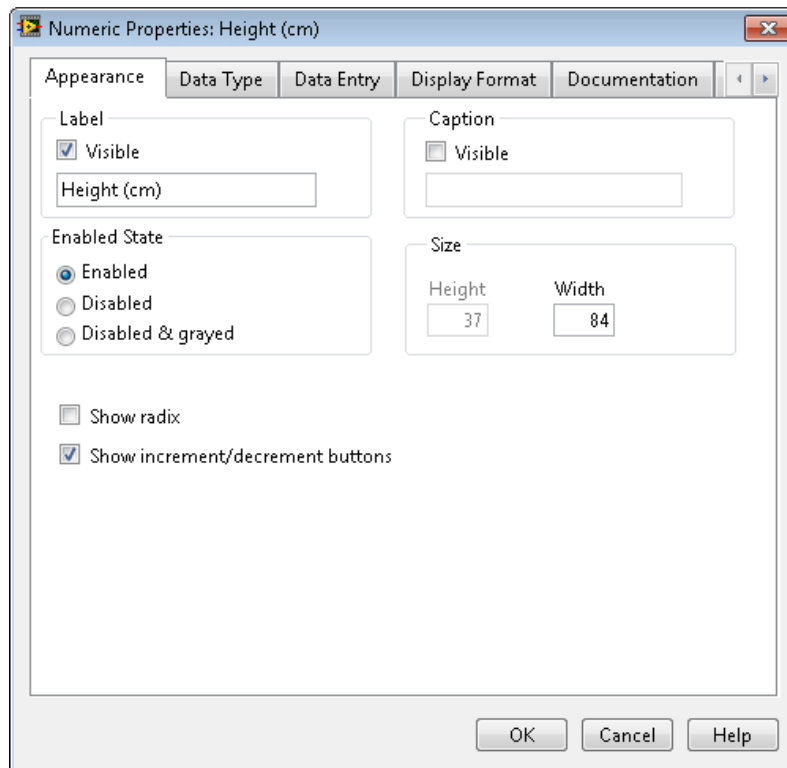
Figure 3-4. Shortcut Menus for Front Panel and Block Diagram Objects



Property Dialog Boxes

Objects also have property dialog boxes that you can use to change the look or behavior of the objects. Right-click an object and select **Properties** from the shortcut menu to access the property dialog box for an object. Figure 3-5 shows the property dialog box for the **Height (cm)** terminal shown in Figure 3-4. The options available on the property dialog box for an object are similar to the options available on the shortcut menu for that object.

Figure 3-5. Property Dialog Box for a Numeric Terminal

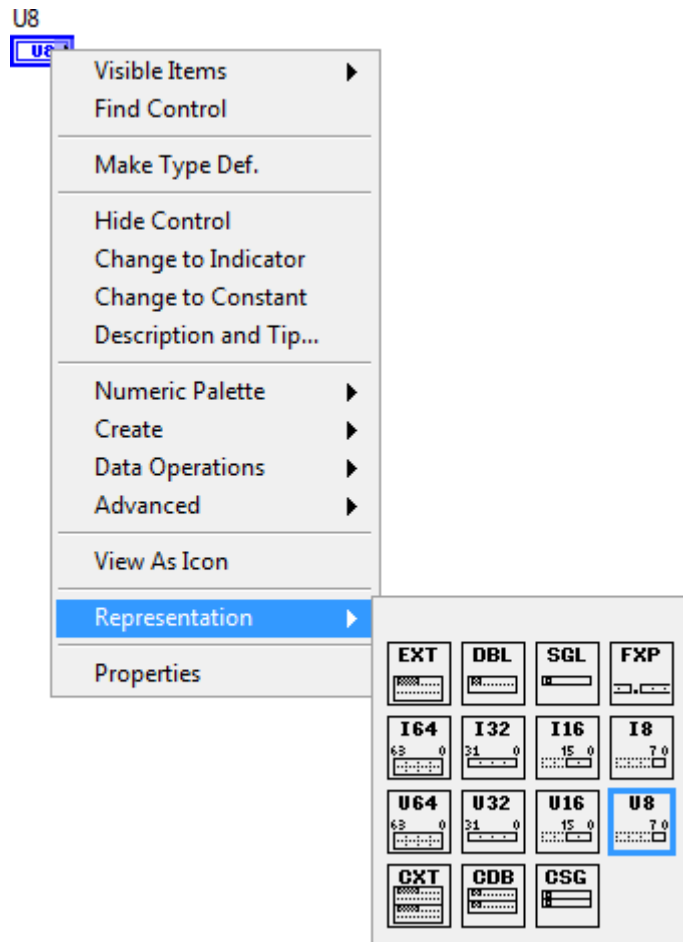


You can select multiple objects on the front panel or the block diagram and edit any properties the objects share. To select multiple objects, use the Positioning tool to drag a selection rectangle around all of the objects you want to edit or hold down the <Shift> key while clicking each object. Right-click an object from the selection and select **Properties** from the shortcut menu to display the **Properties** dialog box. The **Properties** dialog box only displays tabs and properties that the objects you select share. Select similar objects to display more tabs and properties. If you select objects that do not share any common properties, the **Properties** dialog box does not display any tabs or properties.

Numeric Data Types

The numeric data type represents numbers of various types. To change the representation type of a number, right-click the control, indicator, or constant, and select **Representation**, as shown in Figure 3-6.

Figure 3-6. Numeric Representation



When you wire two or more numeric inputs of different representations to a function, the function usually returns the data in the larger, or wider, representation. The functions coerce the smaller representations to the widest representation before execution. LabVIEW places a coercion dot on the terminal where the conversion takes place. Refer to *Numeric Conversion* for more information.

The numeric data type includes the following subcategories of representation—floating-point numbers, signed integers, unsigned integers, and complex numbers.

Floating-Point Numbers

Floating-point numbers represent fractional numbers. In LabVIEW, floating-point numbers are represented with the color orange.

Single-precision (SGL)—Single-precision, floating-point numbers have 32-bit IEEE single-precision format. Use single-precision, floating-point numbers when memory savings are important and you will not overflow the range of the numbers

Double-precision (DBL)—Double-precision, floating-point numbers have 64-bit IEEE double-precision format. Double-precision is the default format for numeric objects. For most situations, use double-precision, floating-point numbers

Extended-precision (EXT)—When you save extended-precision numbers to disk, LabVIEW stores them in a platform-independent 128-bit format. In memory, the size and precision vary depending on the platform. Use extended-precision, floating-point numbers only when necessary. The performance of extended-precision arithmetic vary among platforms.

Fixed-Point Data Type

The fixed-point data type is a numeric data type that represents a set of rational numbers using binary digits, or bits. Unlike the floating-point data type, which allows the total number of bits LabVIEW uses to represent numbers to vary, you can configure fixed-point numbers to always use a specific number of bits. Hardware and targets that only can store and process data with a limited or fixed number of bits then can store and process the numbers. You can specify the range and precision of fixed-point numbers.



Note To represent a rational number using the fixed-point data type, the denominator of the rational number must be a power of 2, because the binary number system is a base-2 number system.

Use the fixed-point data type when you do not need the dynamic functionality of floating-point representation or when you want to work with a target that does not support floating-point arithmetic, such as an FPGA target.

Specify the encoding, word length, and integer word length of a fixed-point number when you want the number to conform to a certain bit size.

Encoding—The binary encoding of the fixed-point number. You can select signed or unsigned. If you select signed, the sign bit is always the first bit in the bit string that represents the data.

Word length—The total number of bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data. LabVIEW accepts a maximum word length of 64 bits. Certain targets might limit data to smaller word lengths. If you open a VI on a target and the VI contains fixed-point data with larger word lengths than the target can accept, the VI contains broken wires. Refer to the documentation for a target to determine the maximum word length the target accepts.

Integer word length—The number of integer bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data, or, given an initial position to the left or right of the most significant bit, the number of bits to shift the binary point to reach the most significant bit. The integer word length can be larger than the word length, and can be positive or negative.

Integers

Integers represent whole numbers. Signed integers can be positive or negative. Use the unsigned integer data types when you know the integer is always positive. In LabVIEW, integers are represented with the color blue.

When LabVIEW converts floating-point numbers to integers, the VI rounds the input to the nearest integer. If the value of the input is midway between two integers, the function returns the nearest even integer.

Byte (I8)—Byte integer numbers have 8 bits of storage and a range of -128 to 127 .

Word (I16)—Word integer numbers have 16 bits of storage and a range of $-32,768$ to $32,767$.

Long (I32)—Long integer numbers have 32 bits of storage and a range of $-2,147,483,648$ to $2,147,483,647$. In most cases, it is best to use a 32-bit integer.

Quad (I64)—Quad integer numbers have 64 bits of storage and a range of $-1e19$ to $1e19$.

Byte (U8)—Byte unsigned integer numbers have 8 bits of storage and a range of 0 to 255 .

Word (U16)—Word unsigned integer numbers have 16 bits of storage and a range of 0 to $65,535$.

Long (U32)—Long unsigned integer numbers have 32 bits of storage and a range of 0 to $4,294,967,295$.

Quad (U64)—Quad unsigned integer numbers have 64 bits of storage and a range of 0 to $2e19$.

Complex Numbers

Complex numbers are concatenated versions of floating-point numbers with a real and an imaginary part. In LabVIEW, because complex numbers are a type of floating-point number, complex numbers are also represented with the color orange. There are three types of complex numbers.

Complex Single (CSG)—Complex single-precision, floating-point numbers consist of real and imaginary values in 32-bit IEEE single-precision format.

Complex Double (CDB)—Complex double-precision, floating-point numbers consist of real and imaginary values in 64-bit IEEE double-precision format.

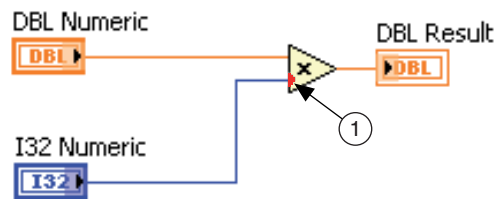
Complex Extended (CXT)—Complex extended-precision, floating-point numbers consist of real and imaginary values in IEEE extended-precision format. In memory, the size and precision of extended-precision numbers vary depending on the platform. In Windows, they have 128-bit IEEE extended-precision format.

Numeric Conversion

LabVIEW can represent numeric data types as signed or unsigned integers, floating-point numeric values, or complex numeric values. Normally, when you wire different representation types to the inputs of a function, the function returns an output in the larger or wider format. If you use a signed integer with an unsigned integer, it will coerce to the unsigned integer. If you use an unsigned integer with a floating point, it will coerce to the floating point. If you use a floating point number with a complex number, it will coerce to the complex number. If you use two numbers of the same type with different bit widths, LabVIEW will coerce to the larger of the two bit widths.

If the number of bits is the same, LabVIEW chooses unsigned over signed integers. For example, if you wire a DBL and an I32 to a Multiply function, the result is a DBL, as shown in Figure 3-7. LabVIEW coerces the 32-bit signed integer because it uses fewer bits than the double-precision, floating-point numeric value. The lower input of the Multiply function shows a red dot, called a coercion dot, that indicates LabVIEW coerced the data.

Figure 3-7. Numeric Conversion Example



1 Coercion Dot

Boolean Values

LabVIEW stores Boolean data as 8-bit values. If the 8-bit value is zero, the Boolean value is FALSE. Any nonzero value represents TRUE. In LabVIEW, the color green represents Boolean data.

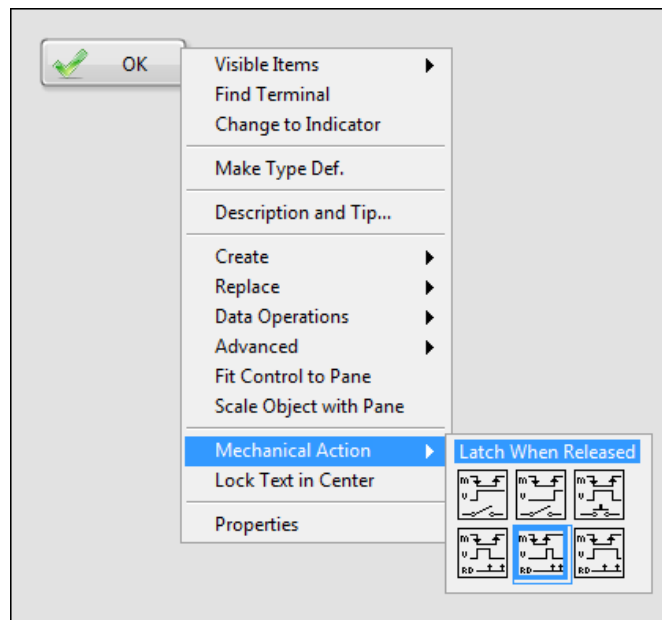
Boolean values also have a mechanical action associated with them. The two major actions are latch and switch. You can select from the following button behaviors:

- **Switch when pressed**—Changes the control value each time you click it with the Operating tool. The frequency with which the VI reads the control does not affect this behavior.
- **Switch when released**—Changes the control value only after you release the mouse button during a mouse click within the graphical boundary of the control. The frequency with which the VI reads the control does not affect this behavior.
- **Switch until released**—Changes the control value when you click it and retains the new value until you release the mouse button. At this time, the control reverts to its default value, similar to the operation of a door buzzer. The frequency with which the VI reads the control does not affect this behavior. You cannot select this behavior for a radio buttons control.
- **Latch when pressed**—Changes the control value when you click it and retains the new value until the VI reads it once. At this point, the control reverts to its default value even if you keep pressing the mouse button. This behavior is similar to a circuit breaker and is useful for stopping a While Loop or for getting the VI to perform an action only once each time you set the control. You cannot select this behavior for a radio buttons control.

- **Latch when released**—Changes the control value only after you release the mouse button within the graphical boundary of the control. When the VI reads it once, the control reverts to its default value. This behavior works in the same manner as dialog box buttons and system buttons. You cannot select this behavior for a radio buttons control.
- **Latch until released**—Changes the control value when you click it and retains the value until the VI reads it once or you release the mouse button, depending on which one occurs last. You cannot select this behavior for a radio buttons control.

To learn more about mechanical action, experiment with the Mechanical Action of Booleans VI in the NI Example Finder.

Figure 3-8. Boolean Mechanical Action



Strings

A string is a sequence of displayable or non-displayable ASCII characters. Strings provide a platform-independent format for information and data. Some of the more common applications of strings include the following:

- Creating simple text messages.
- Controlling instruments by sending text commands to the instrument and returning data values in the form of either ASCII or binary strings which you then convert to numeric values.
- Storing numeric data to disk. To store numeric data in an ASCII file, you must first convert numeric data to strings before writing the data to a disk file.
- Instructing or prompting the user with dialog boxes.

On the front panel, strings appear as tables, text entry boxes, and labels. LabVIEW includes built-in VIs and functions you can use to manipulate strings, including formatting strings, parsing strings, and other editing.

Refer to the *ASCII Codes* topic in the *LabVIEW Help* for more information about ASCII codes and conversion functions.

In LabVIEW, strings are represented with the color pink.

Right-click a string control or indicator on the front panel to select from the display types shown in the Table . The table also shows an example message in each display type.

Table 3-1. String Display Types

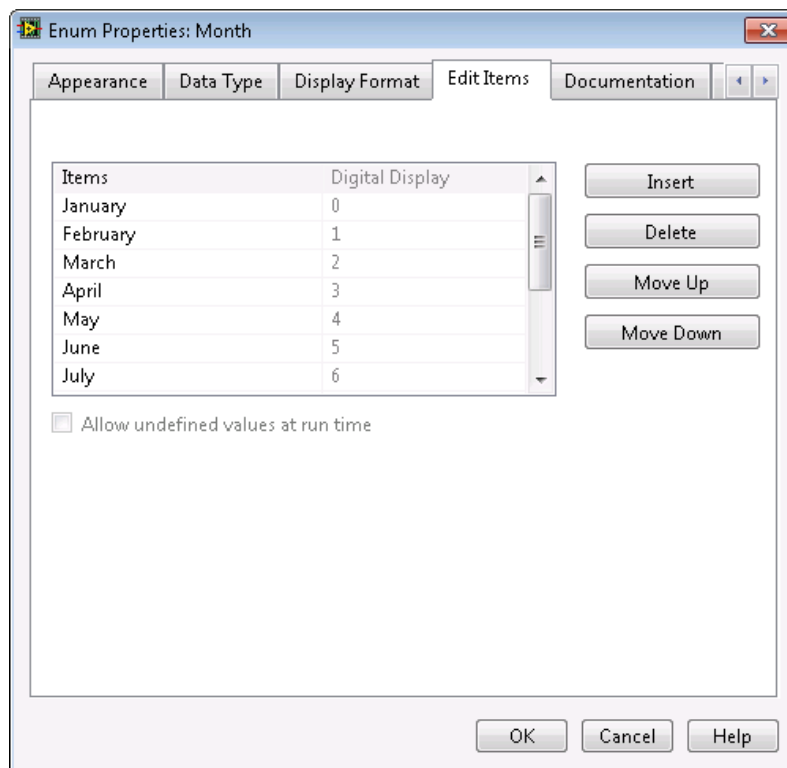
Display Type	Description	Message
Normal Display	Displays printable characters using the font of the control. Non-displayable characters generally appear as boxes.	There are four display types. \ is a backslash.
'\ ' Codes Display	Displays backslash codes for all non-displayable characters.	There\sare\sfour\sdisplay\stypes.\n\\\sis\sasbackslash.
Password Display	Displays an asterisk (*) for each character including spaces.	***** *****
Hex Display	Displays the ASCII value of each character in hex instead of the character itself.	5468 6572 6520 6172 6520 666F 7572 2064 6973 706C 6179 2074 7970 6573 2E0A 5C20 6973 2061 2062 6163 6B73 6C61 7368 2E

LabVIEW stores strings as a pointer to a structure that contains a 4-byte length value followed by a 1D array of byte integers (8-bit characters).

Enums

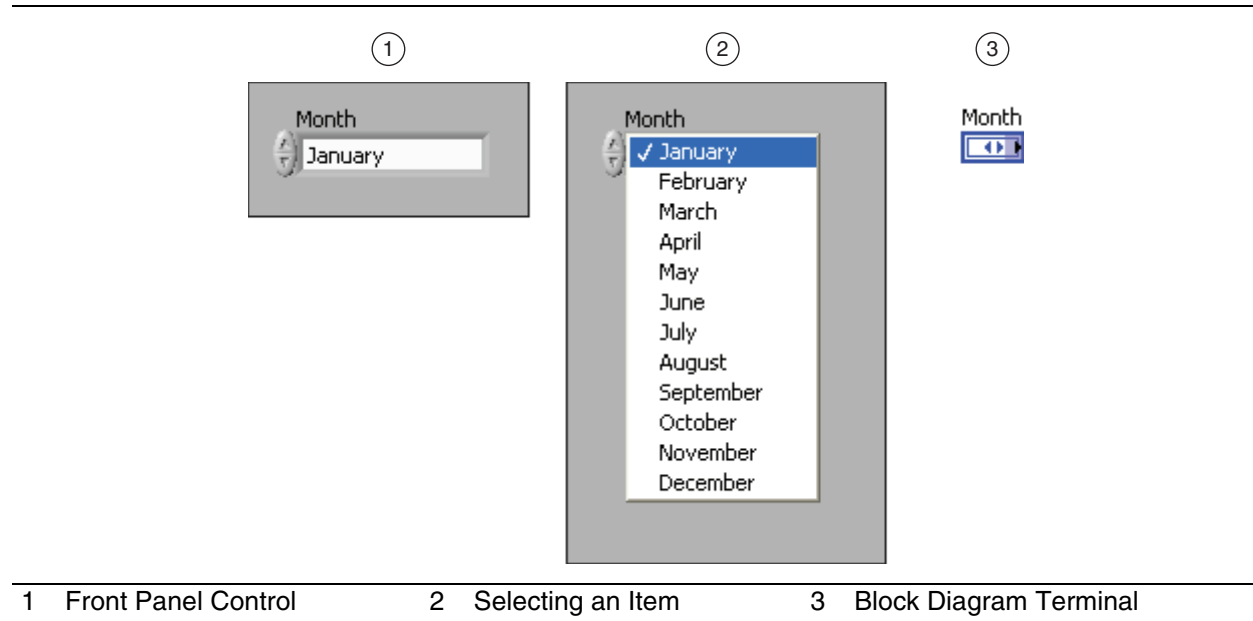
An enum (enumerated control, constant or indicator) is a combination of data types. An enum represents a pair of values, a string and a numeric, where the enum can be one of a list of values. For example, if you created an enum type called Month, the possible value pairs for a Month variable are January-0, February-1, and so on through December-11. Figure 3-9 shows an example of these data pairs in the **Properties** dialog box for an enumerated control.

Figure 3-9. Properties for the Month Enumerated Control



Enums are useful because it is easier to manipulate numbers on the block diagram than strings. Figure 3-10 shows the **Month** enumerated control, the selection of a data pair in the enumerated control, and the corresponding block diagram terminal.

Figure 3-10. Month Enumerated Control



Dynamic

The dynamic data type stores the information generated or acquired by an Express VI. The dynamic data type appears as a dark blue terminal, shown below. Most Express VIs accept and/or return the dynamic data type. You can wire the dynamic data type to any indicator or input that accepts numeric, waveform, or Boolean data. Wire the dynamic data type to an indicator that can best present the data. Indicators include graphs, charts, or numeric indicators.



Most other VIs and functions in LabVIEW do not accept the dynamic data type. To use a built-in VI or function to analyze or process the data the dynamic data type includes, you must convert the dynamic data type.

Use the Convert from Dynamic Data Express VI to convert the dynamic data type to numeric, Boolean, waveform, and array data types for use with other VIs and functions. When you place the Convert from Dynamic Data Express VI on the block diagram, the **Configure Convert from Dynamic Data** dialog box appears. The **Configure Convert from Dynamic Data** dialog box displays options that let you specify how you want to format the data that the Convert from Dynamic Data Express VI returns.



When you wire a dynamic data type to an array indicator, LabVIEW automatically adds the Convert from Dynamic Data Express VI to the block diagram. Double-click the Convert from Dynamic Data Express VI to open the **Configure Convert from Dynamic Data** dialog box to control how the data appears in the array.

C. Documenting Code

Professional developers who maintain and modify VIs know the value of good documentation. Document the block diagram well to ease future modification of the code. In addition, document the front panel window well to explain the purpose of the VI and the front panel objects.

Use tip strips, descriptions, VI Properties, and good design to document front panel windows.

Tip Strips and Descriptions

Tip strips are brief descriptions that appear when you move the cursor over a control or indicator while a VI runs. For example, you might add a tip strip to indicate that a temperature is in degrees Celsius or explain how an input works in an algorithm. Descriptions provide additional information about specific controls and indicators. Descriptions appear in the **Context Help** window when you move the cursor over the object. To add tip strips and descriptions to controls, right-click the control or indicator and select **Description and Tip** from the shortcut menu.

VI Properties

Use the Documentation component of the **VI Properties** dialog box to create VI descriptions and to link from VIs to HTML files or to compiled help files. To display VI Properties right-click the VI icon on the front panel or block diagram and select **VI Properties** from the shortcut menu or select **File»VI Properties**. Then select **Documentation** from the **Categories** drop-down menu. You cannot access this dialog box while a VI runs.

This page includes the following components:

- **VI description**—Contains the text that appears in the **Context Help** window if you move the cursor over the VI icon. Use `` and `` tags around any text in the description you want to format as bold. You also can use the VI Description property to edit the VI description programmatically.
- **Help tag**—Contains the HTML filename or index keyword of the topic you want to link to in a compiled help file. You also can use the Help:Document Tag property to set the help tag programmatically.
- **Help path**—Contains the path to the HTML file or to the compiled help file you want to link to from the Context Help window. If this field is empty, the **Detailed help** link does not appear in the Context Help window, and the **Detailed help** button is dimmed.
- **Browse**—Displays a file dialog box to use to navigate to an HTML file or to a compiled help file to use as the Help path.

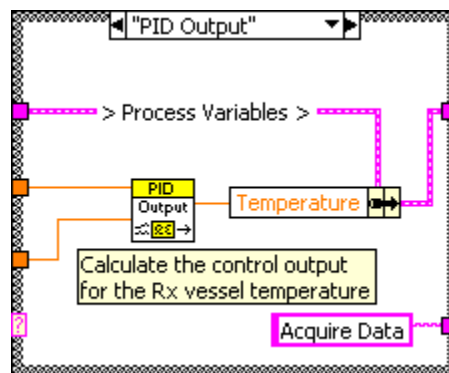
Naming Controls and Indicators

Giving controls and indicators logical and descriptive names adds usability to front panels. For example, if you name a control `Temperature`, a user may not know which units to use. However, naming a control `Temperature °C` adds more information to the front panel. You now know to enter temperatures in metric units.

Graphical Programming

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

Figure 3-11. Documenting a Block Diagram



Use the following guidelines for commenting your VIs:

- Use comments on the block diagram to explain what the code is doing.
- While LabVIEW code can be self-documenting because it is graphical, use labels to describe how the block diagram functions.
- Use labels to identify objects on the front panel and block diagram. LabVIEW includes two different types of labels—owned labels and free labels. Owned labels belong to and move with a particular object and annotate that object only. Free labels are not attached to any object, and you can create, move, rotate or delete them independently.
- Do not show labels on function and subVI calls because they tend to be large and unwieldy. A developer looking at the block diagram can find the name of a function or subVI by using the Context Help window.
- Use owned labels with long wires to identify their use. Labeling wires is useful for wires coming from shift registers and for long wires that span the entire block diagram. Refer to the *Case Structures* section of this lesson for more information about shift registers.
- Label structures to specify the main functionality of the structure.

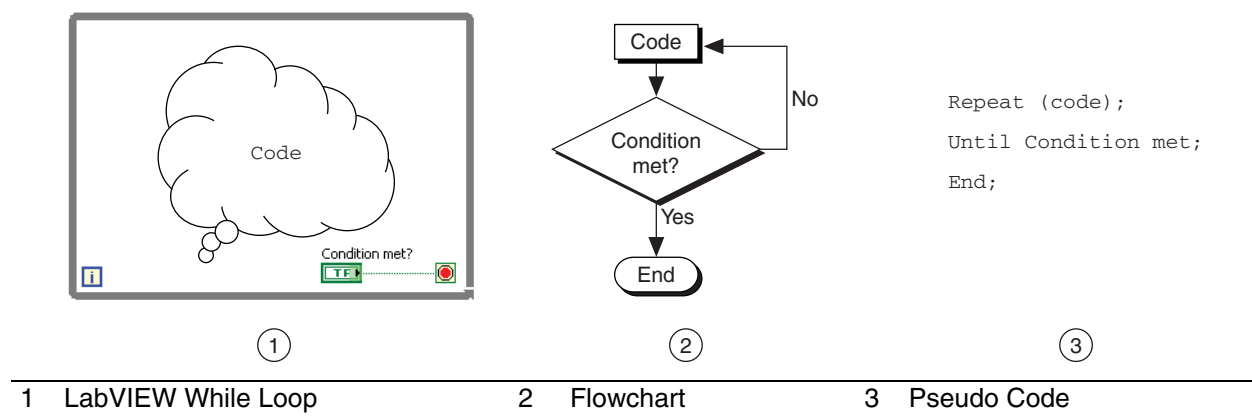
- Label constants to specify the nature of the constant.
- Use free labels to document algorithms that you use on the block diagrams. If you use an algorithm from a book or other reference, provide the reference information.

D. While Loops

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown in Figure 3-12, executes a subdiagram until a condition occurs.

The following illustration shows a While Loop in LabVIEW, a flowchart equivalent of the While Loop functionality, and a pseudo code example of the functionality of the While Loop.

Figure 3-12. While Loop



The While Loop is located on the **Structures** palette. Select the While Loop from the palette then use the cursor to drag a selection rectangle around the section of the block diagram you want to repeat. When you release the mouse button, a While Loop boundary encloses the section you selected.

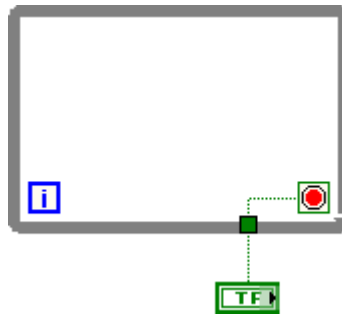
Add block diagram objects to the While Loop by dragging and dropping them inside the While Loop.



Tip The While Loop always executes at least once.

The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value. However, the While Loop does not include a set iteration count and runs infinitely if the condition never occurs.

If a conditional terminal is **Stop if True**, you place the terminal of a Boolean control outside a While Loop, and the control is FALSE when the loop starts, you cause an infinite loop, as shown in the following example. You also cause an infinite loop if the conditional terminal is **Continue if True** and the control outside the loop is set to TRUE, as shown in Figure 3-13.

Figure 3-13. An Infinite Loop

Changing the value of the control does not stop the infinite loop because the value is only read once, before the loop starts. To stop an infinite loop, you must abort the VI by clicking the **Abort Execution** button on the toolbar.

You also can perform basic error handling using the conditional terminal of a While Loop. When you wire an error cluster to the conditional terminal, only the True or False value of the **status** parameter of the error cluster passes to the terminal. Also, the **Stop if True** and **Continue if True** shortcut menu items change to **Stop if Error** and **Continue while Error**.

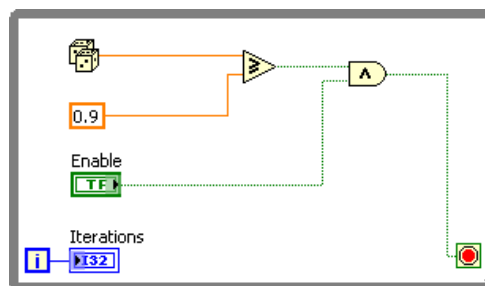
The iteration terminal is an output terminal that contains the number of completed iterations.



The iteration count for the While Loop always starts at zero.

In the following block diagram, the While Loop executes until the Random Number function output is greater than or equal to 0.9 and the **Enable** control is True. The And function returns True only if both inputs are True. Otherwise, it returns False.

In the following example, there is an increased probability of an infinite loop. Generally, the desired behavior is to have one condition met to stop the loop, rather than requiring both conditions to be met.

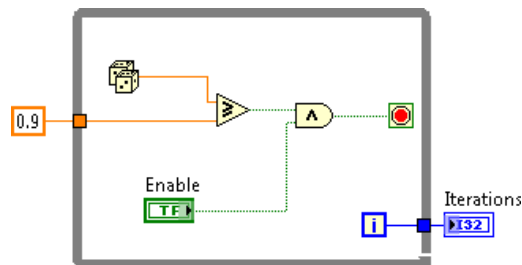
Figure 3-14. Possible Infinite Loop

Tunnels through Structures

Tunnels feed data into and out of structures like While Loops. The tunnel appears as a solid block on the border of the While Loop. The block is the color of the data type wired to the tunnel. Data pass out of a loop after the loop terminates. When a tunnel passes data into a loop, the loop executes only after data arrive at the tunnel.

In the following block diagram, the iteration terminal is connected to a tunnel. The value in the tunnel does not get passed to the **Iterations** indicator until the While Loop finishes executing.

Figure 3-15. While Loop Tunnel



Only the last value of the iteration terminal displays in the **Iterations** indicator.

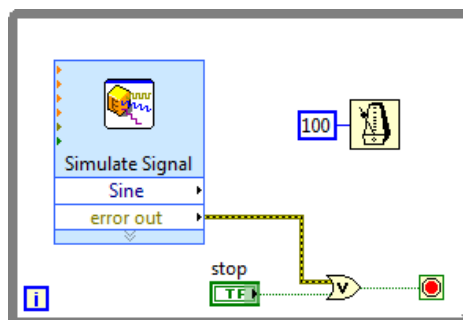
Using While Loops for Error Checking and Error Handling

You can wire an error cluster to the conditional terminal of a While Loop or a For Loop with a conditional terminal to stop the iteration of the loop. If you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the **status** parameter of the error cluster passes to the terminal. If an error occurs, the loop stops.

If you wire an error cluster to the conditional terminal, the shortcut menu items **Stop if True** and **Continue if True** change to **Stop on Error** and **Continue while Error**.

In Figure 3-16, the error cluster and a stop button are used together to determine when to stop the loop. This is the recommended method for stopping most loops.

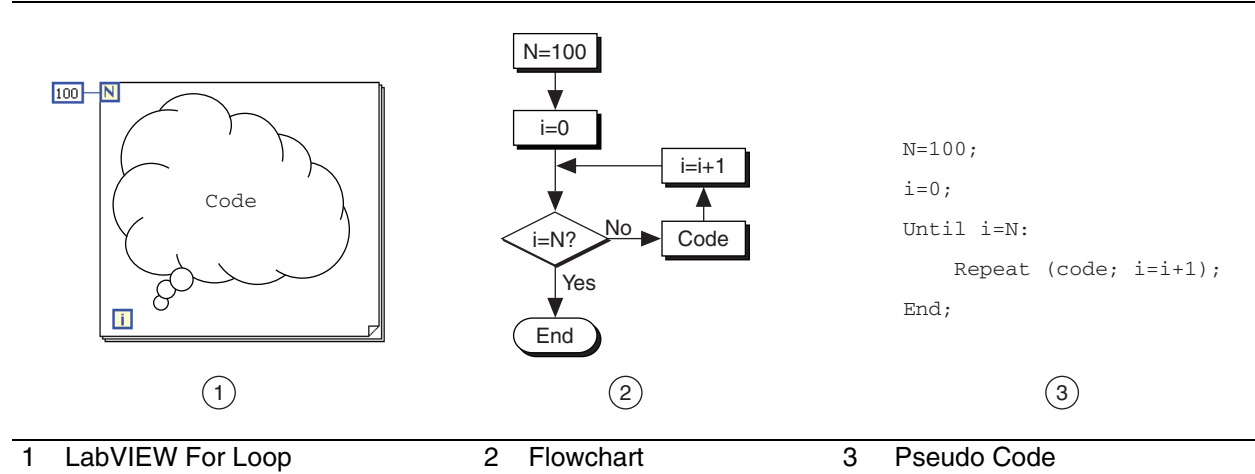
Figure 3-16. Stopping a While Loop



E. For Loops

A For Loop, shown as follows, executes a subdiagram a set number of times. Figure 3-17 shows a For Loop in LabVIEW, a flowchart equivalent of the For Loop functionality, and a pseudo code example of the functionality of the For Loop.

Figure 3-17. For Loop



The For Loop is located on the **Structures** palette. You also can place a While Loop on the block diagram, right-click the border of the While Loop, and select **Replace with For Loop** from the shortcut menu to change a While Loop to a For Loop. The count terminal is an input terminal whose value indicates how many times to repeat the subdiagram.



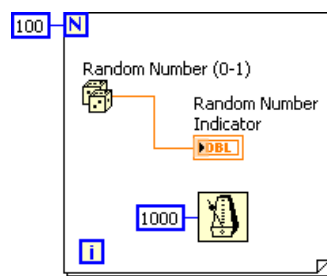
The iteration terminal is an output terminal that contains the number of completed iterations.



The iteration count for the For Loop always starts at zero.

The For Loop in Figure 3-18 generates a random number every second for 100 seconds and displays the random numbers in a numeric indicator.

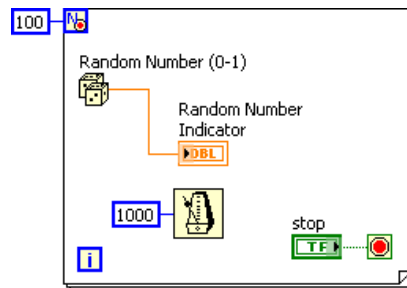
Figure 3-18. For Loop Example



Adding a Conditional Terminal to a For Loop

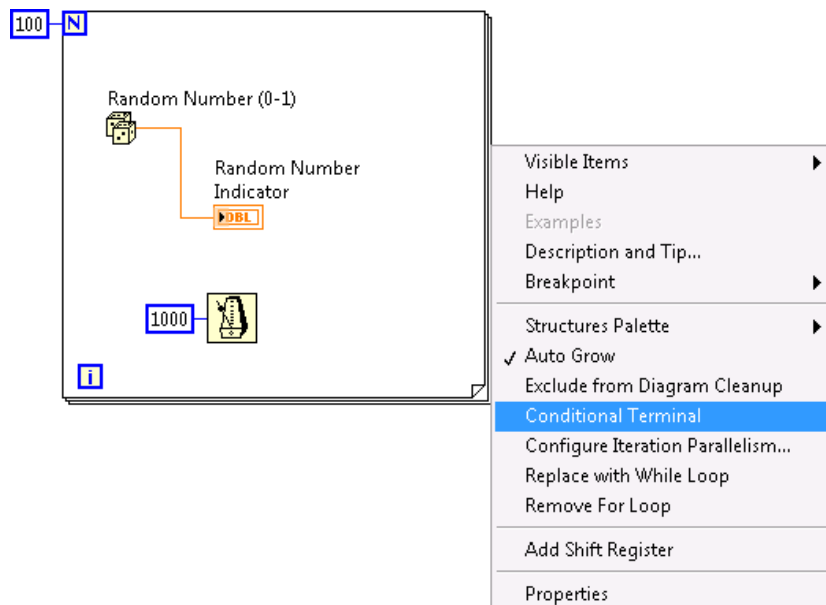
If necessary, you can add a conditional terminal to configure a For Loop to stop when a Boolean condition or an error occurs. A For Loop with a conditional terminal executes until the condition occurs or until all iterations are complete, whichever happens first. For Loops you configure for a conditional exit have a red glyph in the count terminal as well as a conditional terminal in the lower right corner. After you configure the For Loop to exit conditionally, the loop appears similar to Figure 3-19. The following For Loop generates a random number every second until 100 seconds has passed or the user clicks the stop button.

Figure 3-19. For Loop Configured for a Conditional Exit



To add a conditional terminal to a For Loop, right-click on the For Loop border and select Conditional Terminal from the shortcut menu. Then wire the conditional terminal and the count terminal.

Figure 3-20. Adding a Conditional Terminal to a For Loop



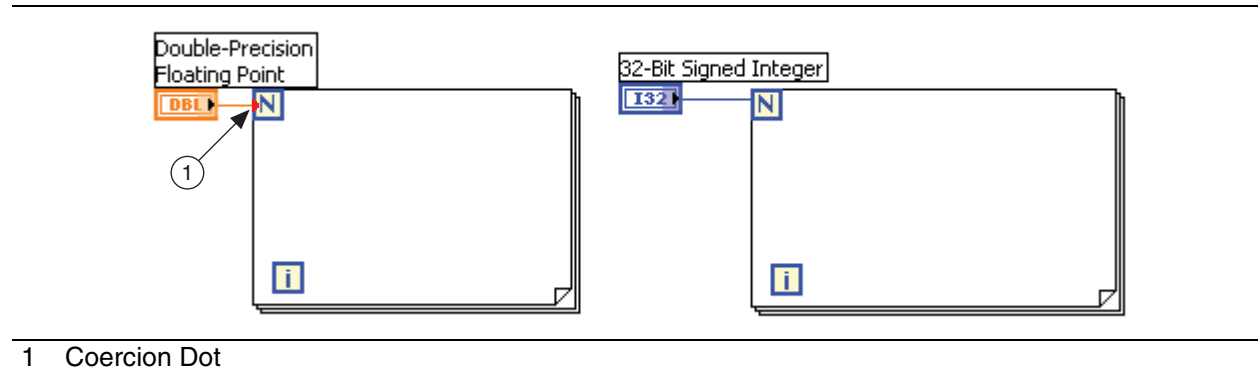
Using For Loops for Error Checking and Error Handling

In a For Loop with a conditional terminal, you also must wire a value to the count terminal or auto-index an input array to set a maximum number of iterations. The For Loop executes until an error occurs or until the number of set iterations completes. Refer to *Using While Loops for Error Checking and Error Handling* for more information.

Numeric Conversion in For Loops

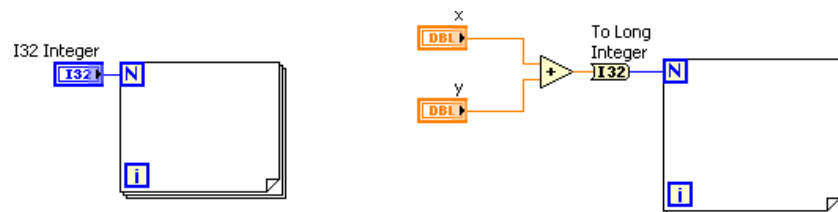
As noted in *Numeric Conversion*, when you wire different data types to the input of a function, the function generally returns the larger or wider data type. However, if you wire a double-precision, floating-point numeric value to the 32-bit count terminal on a For Loop, LabVIEW coerces the larger numeric value to a smaller 32-bit signed integer. Although the conversion is contrary to normal conversion standards, it is necessary, because a For Loop can only execute an integer number of times.

Figure 3-21. Coercion on a For Loop



For better performance, avoid coercion by using matching data types or programmatically converting to matching data types, as shown in Figure 3-22.

Figure 3-22. Avoiding Coercion By Using Matching Data Types



Comparing For Loops and While Loops

For Loops and While Loops differ in several distinct ways. Refer to Table 3-3 for a list of differences between For Loops and While Loops.

Table 3-2. Comparing For Loops and While Loops

For Loop	While Loop
Executes a set number of times unless a conditional terminal is added	Stops executing only if the value at the conditional terminal meets the condition
Can execute zero times	Must execute at least once
Tunnels automatically output an array of data	Tunnels automatically output the last value

F. Timing a VI

When a loop finishes executing an iteration, it immediately begins executing the next iteration, unless it reaches a stop condition. Most often, you need to control the iteration frequency or timing. For example, if you are acquiring data, and you want to acquire the data once every 10 seconds, you need a way to time the loop iterations so they occur once every 10 seconds.

Even if you do not need the execution to occur at a certain frequency, you need to provide the processor with time to complete other tasks, such as responding to the user interface. This section introduces some methods for timing your loops.

Wait Functions

Place a wait function inside a loop to allow a VI to sleep for a set amount of time. This allows your processor to address other tasks during the wait time. Wait functions use the millisecond clock of the operating system.

The Wait Until Next ms Multiple function monitors a millisecond counter and waits until the millisecond counter reaches a multiple of the amount you specify. Use this function to synchronize activities. Place this function in a loop to control the loop execution rate. For this function to be effective, your code execution time must be less than the time specified for this function. The execution rate for the first iteration of the loop is indeterminate.



The Wait (ms) function waits until the millisecond counter counts to an amount equal to the input you specify. This function guarantees that the loop execution rate is at least the amount of the input you specify.





Note The Time Delay Express VI behaves similarly to the Wait (ms) function with the addition of built-in error clusters. Refer to Lesson 2, *Troubleshooting and Debugging VIs*, for more information about error clusters.

Elapsed Time

In some cases, it is useful to determine how much time elapses after some point in your VI. The Elapsed Time Express VI indicates the amount of time that elapses after the specified start time. This Express VI keeps track of time while the VI continues to execute. This Express VI does not provide the processor with time to complete other tasks. You will use the Elapsed Time Express VI in the Weather Station course project.



G. Data Feedback in Loops

When programming with loops, you often must access data from previous iterations of the loop in LabVIEW. For example, if you are acquiring one piece of data in each iteration of a loop and must average every five pieces of data, you must retain the data from previous iterations of the loop.



Note Feedback Nodes are another method for retaining information from a previous iteration. Refer to the *Feedback Node* topic of the *LabVIEW Help* for more information about Feedback Nodes.

Shift Registers

Shift registers are similar to static variables in text-based programming languages.

Use shift registers when you want to pass values from previous iterations through the loop to the next iteration. A shift register appears as a pair of terminals directly opposite each other on the vertical sides of the loop border.



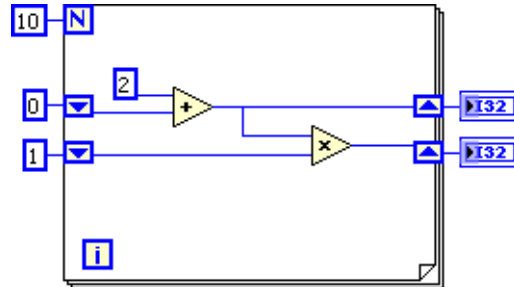
The terminal on the right side of the loop contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data connected to the right side of the register to the next iteration. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register.

Create a shift register by right-clicking the left or right border of a loop and selecting **Add Shift Register** from the shortcut menu.

A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type.

You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within your loop, use multiple shift registers to store the data values from those different processes in the structure, as shown in the following figure.

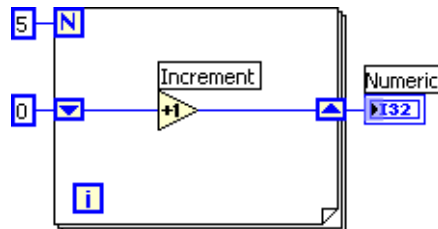
Figure 3-23. Using Multiple Shift Registers



Initializing Shift Registers

Initializing a shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop, as shown in Figure 3-24.

Figure 3-24. Initialized Shift Register

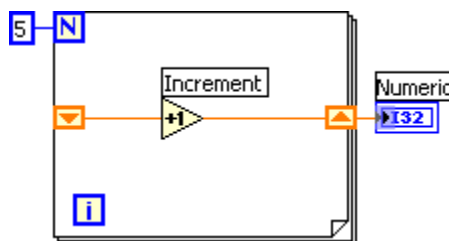


In Figure 3-24, the For Loop executes five times, incrementing the value the shift register carries by one each time. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator and the VI quits. Each time you run the VI, the shift register begins with a value of 0.

If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or, if the loop has never executed, the default value for the data type.

Use an uninitialized shift register to preserve state information between subsequent executions of a VI. Figure 3-25 shows an uninitialized shift register.

Figure 3-25. Uninitialized Shift Register



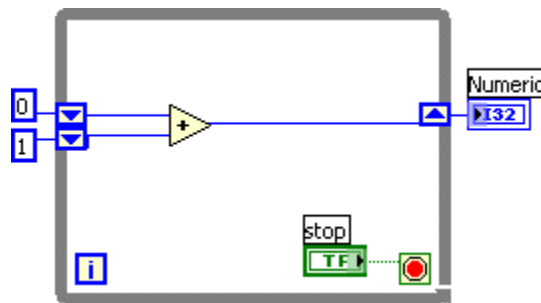
In Figure 3-25, the For Loop executes five times, incrementing the value the shift register carries by one each time. The first time you run the VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator, and the VI quits. The next time you run the VI, the shift register begins with a value of 5, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value, 10, to the indicator. If you run the VI again, the shift register begins with a value of 10, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI.

Stacked Shift Registers

Stacked shift registers let you access data from previous loop iterations. Stacked shift registers remember values from multiple previous iterations and carry those values to the next iterations. To create a stacked shift register, right-click the left terminal and select **Add Element** from the shortcut menu.

Stacked shift registers can occur only on the left side of the loop because the right terminal transfers the data generated from only the current iteration to the next iteration, as shown in Figure 3-26.

Figure 3-26. Using Stacked Shift Registers



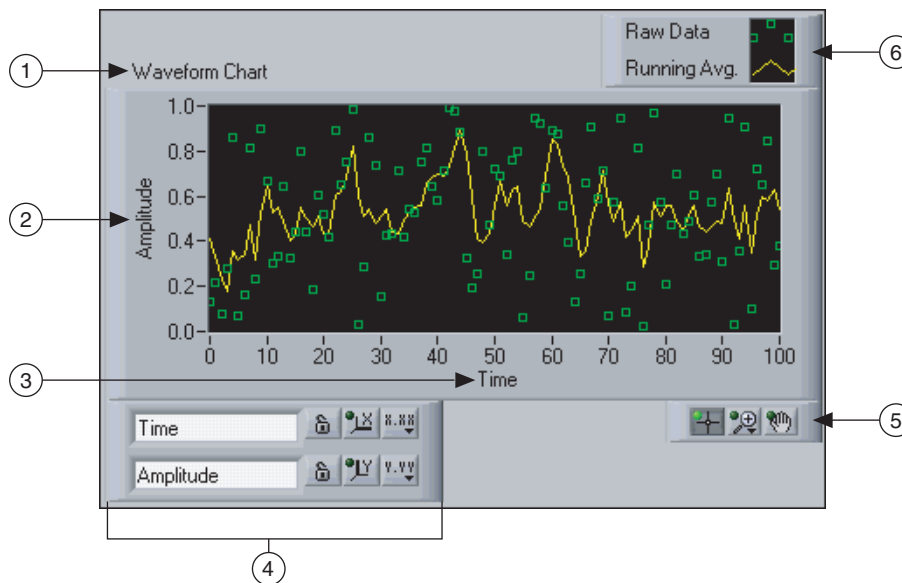
If you add another element to the left terminal in the previous block diagram, values from the last two iterations carry over to the next iteration, with the most recent iteration value stored in the top shift register. The bottom terminal stores the data passed to it from the previous iteration.

H. Plotting Data – Waveform Chart

The waveform chart is a special type of numeric indicator that displays one or more plots of data typically acquired at a constant rate. Waveform charts can display single or multiple plots.

Figure 3-27 shows the elements of a multiplot waveform chart. Two plots are displayed: Raw Data and Running Avg.

Figure 3-27. Elements of a Waveform Charts



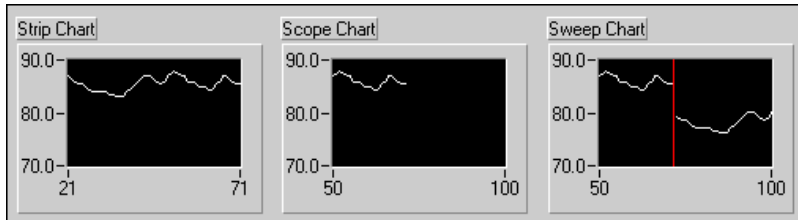
1 Label	3 X-scale	5 Graph Palette
2 Y-scale	4 Scale Legend	6 Plot Legend

Configure how the chart updates to display new data. Right-click the chart and select **Advanced» Update Mode** from the shortcut menu to set the chart update mode. The chart uses the following modes to display data:

- **Strip Chart**—Shows running data continuously scrolling from left to right across the chart with old data on the left and new data on the right. A strip chart is similar to a paper tape strip chart recorder. **Strip Chart** is the default update mode.
- **Scope Chart**—Shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to right. For each new value, the chart plots the value to the right of the last value. When the plot reaches the right border of the plotting area, LabVIEW erases the plot and begins plotting again from the left border. The retracing display of a scope chart is similar to an oscilloscope.
- **Sweep Chart**—Works similarly to a scope chart except it shows the old data on the right and the new data on the left separated by a vertical line. LabVIEW does not erase the plot in a sweep chart when the plot reaches the right border of the plotting area. A sweep chart is similar to an EKG display.

Figure 3-28 shows an example of each chart update mode. The scope chart and sweep chart have retracing displays similar to an oscilloscope. Because retracing a plot requires less overhead, the scope chart and the sweep chart display plots significantly faster than the strip chart.

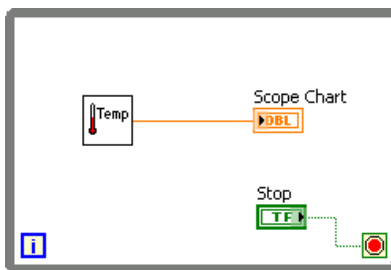
Figure 3-28. Chart Update Modes



Wiring Charts

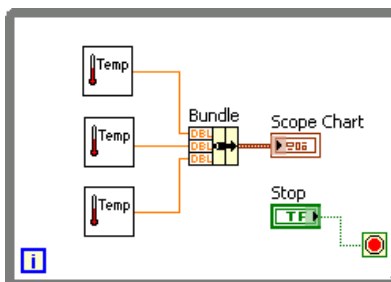
You can wire a scalar output directly to a waveform chart. The waveform chart terminal shown in Figure 3-29 matches the input data type.

Figure 3-29. Wiring a Single Plot to a Waveform Chart



Waveform charts can display multiple plots together using the Bundle function located on the **Cluster, Class & Variant** palette. In Figure 3-30, the Bundle function bundles the outputs of the three VIs to plot on the waveform chart.

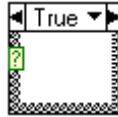
Figure 3-30. Wiring Multiple Plots to a Waveform Chart



The waveform chart terminal changes to match the output of the Bundle function. To add more plots, use the Positioning tool to resize the Bundle function.

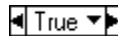
I. Case Structures

A Case structure has two or more subdiagrams, or cases.



Only one subdiagram is visible at a time, and the structure executes only one case at a time. An input value determines which subdiagram executes. The Case structure is similar to switch statements or if...then...else statements in text-based programming languages.

The case selector label at the top of the Case structure contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side.



Click the decrement and increment arrows to scroll through the available cases. You also can click the down arrow next to the case name and select a case from the pull-down menu.

Wire an input value, or selector, to the selector terminal to determine which case executes.



You must wire an integer, Boolean value, string, or enumerated type value to the selector terminal. You can position the selector terminal anywhere on the left border of the Case structure. If the data type of the selector terminal is Boolean, the structure has a True case and a False case. If the selector terminal is an integer, string, or enumerated type value, the structure can have any number of cases.



Note By default, string values you wire to the selector terminal are case sensitive. To allow case-insensitive matches, wire a string value to the selector terminal, right-click the border of the Case structure, and select **Case Insensitive Match** from the shortcut menu.

If you do not specify a default case for the Case structure to handle out-of-range values, you must explicitly list every possible input value. For example, if the selector is an integer and you specify cases for 1, 2, and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.



Note You cannot specify a default case if you wire a Boolean control to the selector. If you right-click the case selector label, **Make This The Default Case** does not appear in the shortcut menu. Make the Boolean control TRUE or FALSE to determine which case to execute.

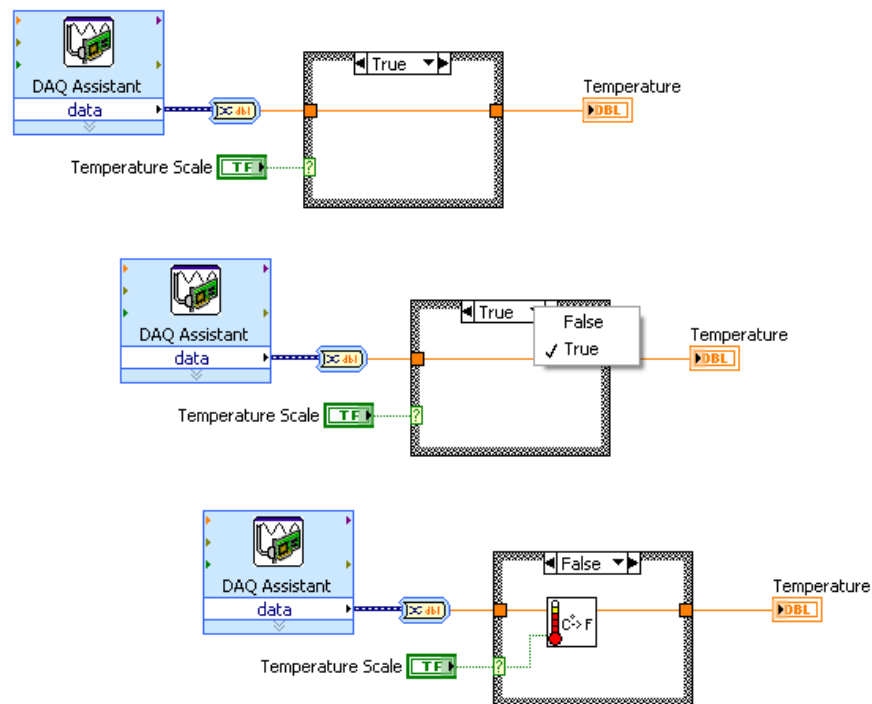
To convert a Case structure to a Stacked Sequence structure, right-click the Case structure and select **Replace with Stacked Sequence** from the shortcut menu.

Right-click the Case structure border to add, duplicate, remove, or rearrange cases, and to select a default case.

Selecting a Case

Figure 3-31 shows a VI that uses a Case structure to execute different code dependent on whether a user selects °C or °F for temperature units. The top block diagram shows the True case in the foreground. In the middle block diagram, the False case is selected. To select a case, enter the value in the case selector identifier or use the Labeling tool to edit the values. After you select another case, that case displays on the block diagram, as shown in the bottom block diagram of Figure 3-31.

Figure 3-31. Changing the Case View of a Case Structure



If you enter a selector value that is not the same type as the object wired to the selector terminal, the value appears red. This indicates that the VI will not run until you delete or edit the value. Also, because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest integer. If you type a floating-point value in the case selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

Input and Output Tunnels on Case Structures

You can create multiple input and output tunnels for a Case structure. Inputs are available to all cases, but cases do not need to use each input. However, you must define an output tunnel for each case.

Consider the following example: a Case structure on the block diagram has an output tunnel, but in at least one of the cases, there is no output value wired to the tunnel. If you run this case, LabVIEW does not know what value to return for the output. LabVIEW indicates this error by leaving the center of the tunnel white. The unwired case might not be the case that is currently visible on the block diagram.

To correct this error, display the case(s) that contain(s) the unwired output tunnel and wire an output to the tunnel. You also can right-click the output tunnel and select **Use Default If Unwired** from the shortcut menu to use the default value for the tunnel data type for all unwired tunnels. When the output is wired in all cases, the output tunnel is a solid color.

Avoid using the **Use Default If Unwired** option. Using this option does not document the block diagram well, and can confuse other programmers using your code. The **Use Default If Unwired** option also makes debugging code difficult. If you use this option, be aware that the default value used is the default value for the data type that is wired to the tunnel. For example, if the tunnel is a Boolean data type, the default value is `FALSE`. Refer to Table 3-3 for a list of default values for data types.

Table 3-3. Data Type Default Values

Data Type	Default Value
Numeric	0
Boolean	FALSE
String	empty (" ")

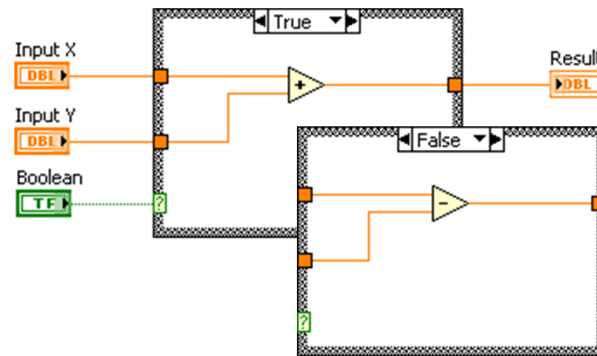
Examples

In the following examples, the numeric values pass through tunnels to the Case structure and are either added or subtracted, depending on the value wired to the selector terminal.

Boolean Case Structure

Figure 3-32 shows a Boolean Case structure. The cases overlap each other to simplify the illustration.

Figure 3-32. Boolean Case Structure



If the Boolean control wired to the selector terminal is True, the VI adds the numeric values. Otherwise, the VI subtracts the numeric values.

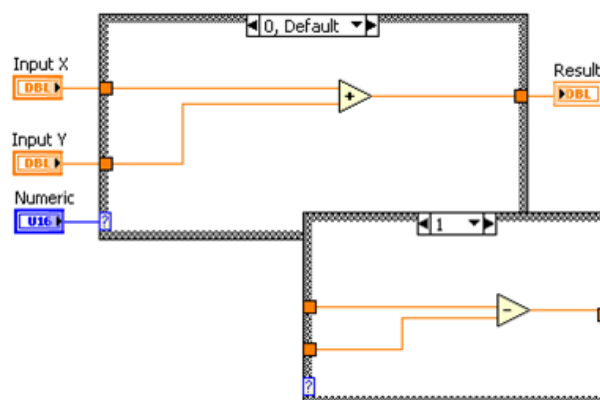
In situations where you are simply choosing between two values based on a boolean input, a Select statement can be used rather than a Boolean Case structure.



Integer Case Structure

Figure 3-33 shows an integer Case structure.

Figure 3-33. Integer Case Structure

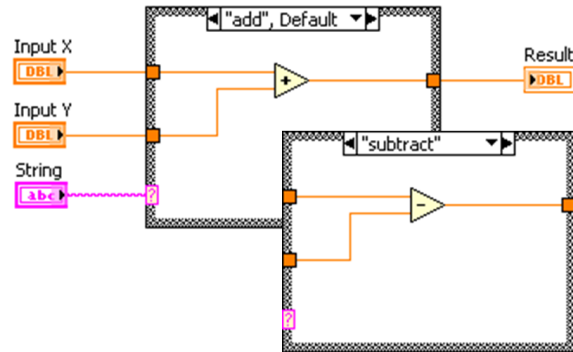


Integer is a text ring control located on the **Text Controls** palette that associates numeric values with text items. If the Integer wired to the selector terminal is 0 (add), the VI adds the numeric values. If the value is 1 (subtract), the VI subtracts the numeric values. If Integer is any other value than 0 (add) or 1 (subtract), the VI adds the numeric values, because that is the default case.

String Case Structure

Figure 3-34 shows a string Case structure.

Figure 3-34. String Case Structure

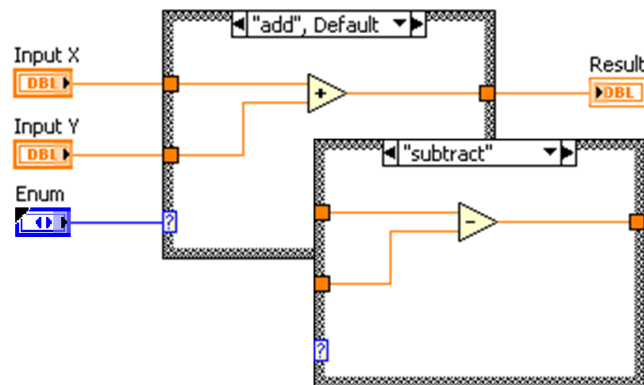


If **String** is `add`, the VI adds the numeric values. If **String** is `subtract`, the VI subtracts the numeric values.

Enumerated Case Structure

Figure 3-35 shows an enumerated Case structure.

Figure 3-35. Enumerated Case Structure



An enumerated type control gives users a list of items from which to select. The data type of an enumerated type control includes information about the numeric values and string labels in the control. The case selector displays the string label for each item in the enumerated type control when you select **Add Case For Every Value** from the Case structure shortcut menu. The Case structure executes the appropriate case subdiagram based on the current item in the enumerated type control. In the previous block diagram, if **Enum** is `add`, the VI adds the numeric values. If **Enum** is `subtract`, the VI subtracts the numeric values.

Using Case Structures for Error Handling

The following example shows a Case structure where an error cluster defines the cases.

Figure 3-36. No Error Case

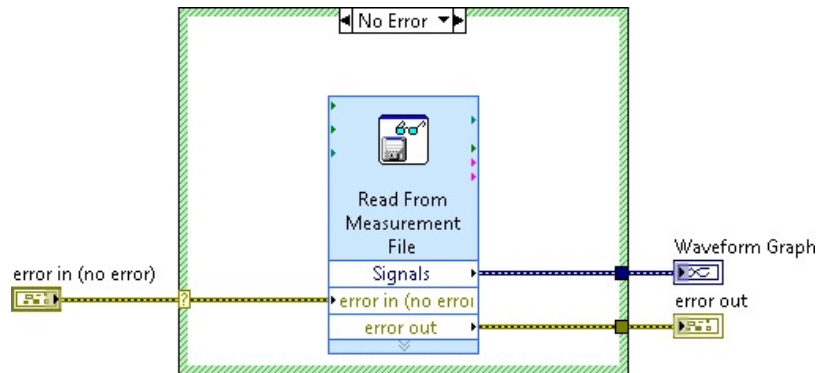
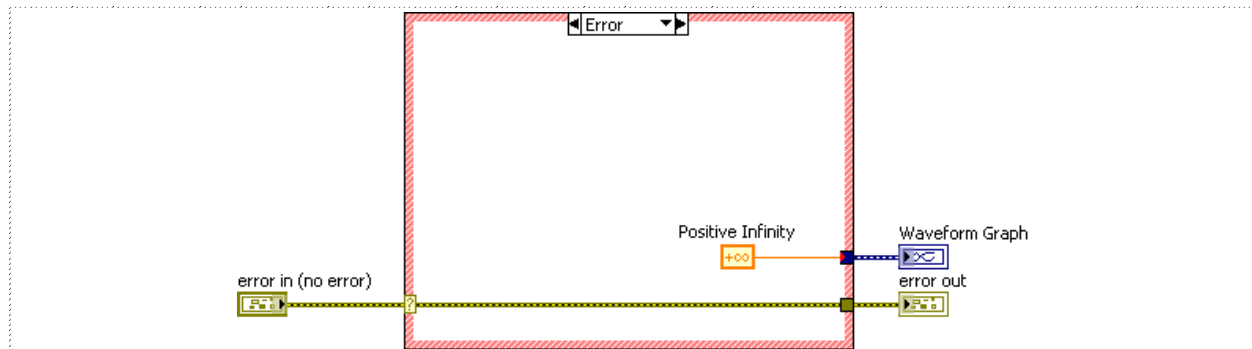


Figure 3-37. Error Case



When you wire an error cluster to the selector terminal of a Case structure, the case selector label displays two cases—Error and No Error—and the border of the Case structure changes color—red for Error and green for No Error. If an error occurs, the Case structure executes the Error subdiagram.

When you wire an error cluster to the selection terminal, the Case structure recognizes only the **status** Boolean element of the cluster.

Self-Review: Quiz

1. If an input to a function is marked with a red dot (known as a coercion dot), what does the dot indicate?
 - a. Data was transferred into a structure.
 - b. A For Loop was configured with a conditional terminal.
 - c. A For Loop iteration terminal is unwired.
 - d. The value passed into a node was converted to a different representation.

2. Which structure must run at least one time?
 - a. While Loop
 - b. For Loop

3. Which is *only* available on the block diagram?
 - a. Control
 - b. Constant
 - c. Indicator
 - d. Connector pane

4. Which mechanical action causes a Boolean control in the FALSE state to change to TRUE when you click it and stay TRUE until LabVIEW has read the value?
 - a. Switch until released
 - b. Switch when released
 - c. Latch until released
 - d. Latch when released

Self-Review: Quiz Answers

1. If an input to a function is marked with a red dot (known as a coercion dot), what does the dot indicate?
 - a. Data was transferred into a structure.
 - b. A For Loop was configured with a conditional terminal.
 - c. A For Loop iteration terminal is unwired.
 - d. The value passed into a node was converted to a different representation.**

2. Which structure must run at least one time?
 - a. While Loop**
 - b. For Loop

3. Which is *only* available on the block diagram?
 - a. Control
 - b. Constant**
 - c. Indicator
 - d. Connector pane

4. Which mechanical action causes a Boolean control in the FALSE state to change to TRUE when you click it and stay TRUE until you release it and LabVIEW has read the value?
 - a. Switch until released
 - b. Switch when released
 - c. Latch until released
 - d. Latch when released**

Notes

Developing Modular Applications

This lesson describes how to develop modular applications. The power of LabVIEW lies in the hierarchical nature of the VI. After you create a VI, you can use it on the block diagram of another VI. There is no limit on the number of layers in the hierarchy. Using modular programming helps you manage changes and debug the block diagram quickly.

Topics

- A. Understanding Modularity
- B. Building the Icon and Connector Pane
- C. Using SubVIs

A. Understanding Modularity

Modularity defines the degree to which a program is composed of discrete modules such that a change to one module has minimal impact on other modules. Modules in LabVIEW are called subVIs.

A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages. When you double-click a subVI, a front panel and block diagram appear, rather than a dialog box in which you can configure options. The front panel includes controls and indicators. The block diagram includes wires, front panel icons, functions, possibly subVIs, and other LabVIEW objects that also might look familiar.

The upper right corner of the front panel window and block diagram window displays the icon for the VI. This icon is the same as the icon that appears when you place the VI on the block diagram.

As you create VIs, you might find that you perform a certain operation frequently. Consider using subVIs or loops to perform that operation repetitively. For example, the following block diagram contains two identical operations.

Figure 4-1. Block Diagram with Two Identical Operations

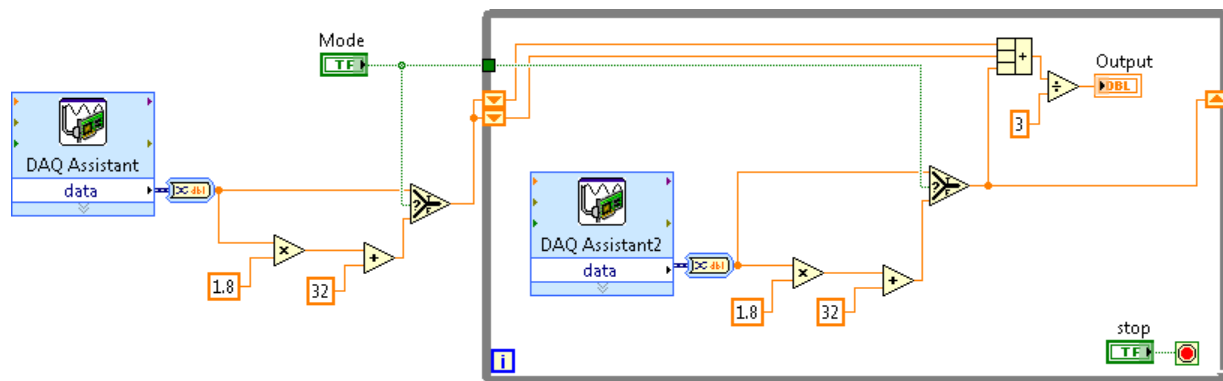
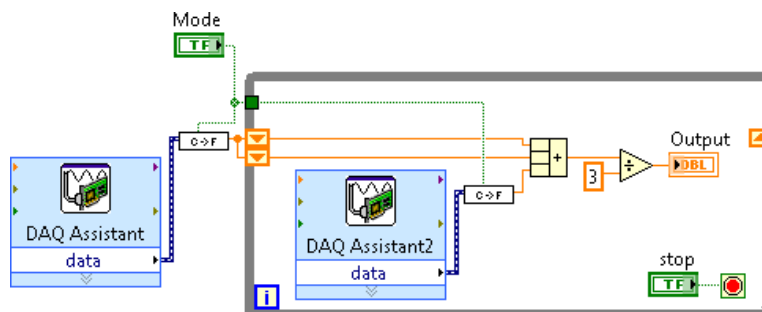
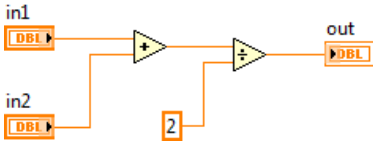
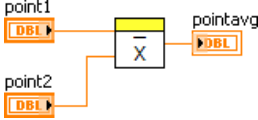


Figure 4-2. Block Diagram with SubVIs for Identical Operations



The example calls the Temperature VI as a subVI twice on its block diagram and functions the same as the previous block diagram. You also can reuse the subVI in other VIs.

The following pseudo-code and block diagrams demonstrate the analogy between subVIs and subroutines.

Function Code	Calling Program Code
<pre>function average (in1, in2, out) { out = (in1 + in2)/2.0; }</pre>	<pre>main { average (point1, point2, pointavg) }</pre>
SubVI Block Diagram	Calling VI Block Diagram
	

B. Building the Icon and Connector Pane

After you build a VI front panel and block diagram, build the icon and the connector pane so you can use the VI as a subVI. The icon and connector pane correspond to the function prototype in text-based programming languages. Every VI displays an icon in the upper right corner of the front panel and block diagram windows.



A VI icon is a graphical representation of a VI. It can contain text, images, or a combination of both. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. If you add the VI to a palette, the VI icon also appears on the **Functions** palette. You can double-click the icon in the front panel window or block diagram window to customize or edit it.



Note Customizing the icon is recommended, but optional. Using the default LabVIEW icon does not affect functionality.

You also need to build a connector pane to use the VI as a subVI.



The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls and receives the results at its output terminals from the front panel indicators.

Creating an Icon

Icons are graphical representations of VIs.

Every VI displays an icon in the upper right corner of the front panel and block diagram windows.



The default VI icon contains a number that indicates how many new VIs, up to nine VIs, you have opened since launching LabVIEW. To disable this numbering, select **Tools»Options»Front Panel** and remove the checkmark from the **Use numbers in icons of new VIs (1 through 9)** checkbox.

An icon can contain text or images. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. If you add the VI to a palette, the VI icon also appears on the **Functions** palette.

Use the **Icon Editor** dialog box to edit a VI icon. Double-click the icon in the upper right corner of the front panel or block diagram window to display the **Icon Editor** dialog box.

Create an icon to represent a VI or custom control graphically. Use the **Icon Editor** dialog box to create or edit icons.

You can use banners to identify related VIs. National Instruments recommends creating and saving a banner as a template. You then can use this template for a related VI icon and modify the body of the VI icon to provide information about the specific VI.

Saving a Banner as a Template

Complete the following steps to save a banner as an icon template for a VI.

1. Double-click the icon in the upper right corner of the front panel window or block diagram window, or right-click the icon and select **Edit Icon** from the shortcut menu, to display the **Icon Editor** dialog box.
2. Press the <Ctrl-A> keys to select all user layers of the icon, and press the <Delete> key to delete the selection. The default icon is a single user layer called **VI Icon**.
3. On the **Templates** page, select the `_blank.png` icon template from the **VI»Frameworks** category. You can browse templates by category or by keyword.
4. Use the Fill tool on the right side of the **Icon Editor** dialog box to fill the banner of the icon with a color.
5. Use the Text tool to enter text in the banner of the icon. While the text is active, you can move the text by pressing the arrow keys.
6. Select **File»Save As»Template** to display the **Save Icon As** dialog box and save the icon as a template for later use. LabVIEW saves icon templates as 256-color .png files.

Creating a VI Icon from a Template

Complete the following steps to create a VI icon that uses the template you created.

1. Press the <Ctrl-A> keys to select all user layers of the icon, and press the <Delete> key to delete the selection.
2. On the **Templates** page, select the template you created. You can browse templates by category or by keyword.
3. On the **Icon Text** page, enter up to four lines of icon text for the body of the icon. You can configure the font, alignment, size, and color of the text. If you place a checkmark in the **Center text vertically** checkbox, the **Icon Editor** dialog box centers the icon text vertically within the body of the icon.
4. On the **Glyphs** page, drag and drop glyphs onto the **Preview** area.
Press the <F> key or the <R> key to flip a glyph horizontally or rotate a glyph clockwise, respectively, as you move the glyph. You also can double-click a glyph to place the glyph in the top-left corner of the icon. You can browse glyphs by category or by keyword.
5. Use the Move tool to move any glyph. Each glyph is on a separate layer and therefore moves separately. Notice that the rest of the icon becomes dimmed when you select a glyph so you can identify which selection you are moving.
6. Use the editing tools on the right side of the **Icon Editor** dialog box to edit the icon further if necessary.

The **Icon Editor** dialog box creates a new user layer for each non-consecutive use of the editing tools. Select **Layers»Create New Layer** to create a new user layer during consecutive uses of the editing tools.



Note You cannot modify the icon template or icon text with the editing tools on the right side of the **Icon Editor** dialog box. Use the **Templates** page and the **Icon Text** page to modify the icon template and icon text, respectively.

7. (Optional) Select **Layers»Show Layers Page** to display the **Layers** page. Use this page to configure the name, opacity, visibility, and order of the layers of the icon.
8. Click the **OK** button to save the icon information with the VI and close the **Icon Editor** dialog box.

You also can drag a graphic from anywhere in the file system and drop it in the upper right corner of the front panel window to use the graphic as a VI icon. You can drag and drop .png, .bmp, or .jpg files.



Note If you modify an icon by dragging and dropping a graphic from the file system, LabVIEW creates a user layer called VI Icon for the graphic and deletes any other existing icon information from the **Icon Editor** dialog box.

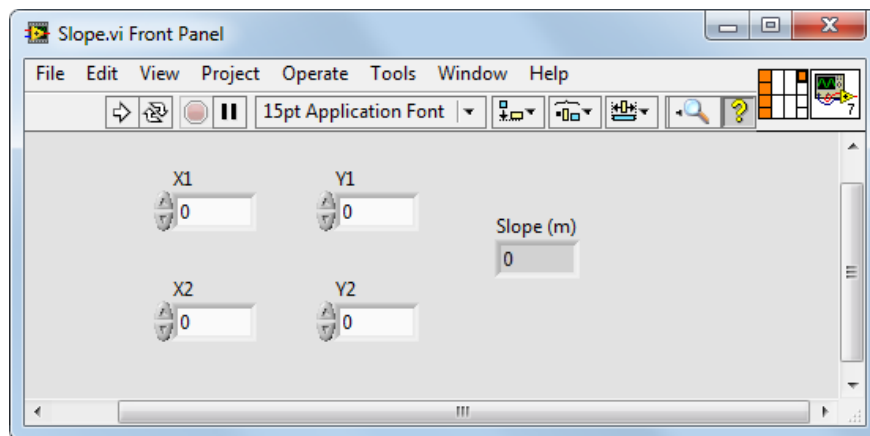
Setting up the Connector Pane

Define connections by assigning a front panel control or indicator to each of the connector pane terminals. The connector pane is located to the left of the VI icon in the upper right corner of the front panel window. When you open LabVIEW, you see a default connector pattern.

Each rectangle on the connector pane represents a terminal. Use the rectangles to assign inputs and outputs. The default connector pane pattern is $4 \times 2 \times 2 \times 4$. You can select a different pattern by right-clicking the connector pane and selecting **Patterns** from the shortcut menu. If you anticipate changes to the VI that would require a new input or output, keep the default connector pane pattern to leave extra terminals unassigned.

The front panel in Figure 4-3 has four controls and one indicator, so LabVIEW displays four input terminals and one output terminal on the connector pane.

Figure 4-3. Slope VI Front Panel



Selecting and Modifying Terminal Patterns

Select a different terminal pattern for a VI by right-clicking the connector pane and selecting **Patterns** from the shortcut menu. For example, you can select a connector pane pattern with extra terminals. You can leave the extra terminals unconnected until you need them. This flexibility enables you to make changes with minimal effect on the hierarchy of the VIs.

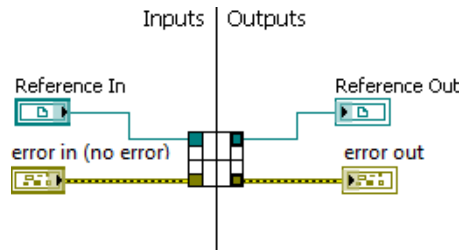
You also can have more front panel controls or indicators than terminals. You can assign up to 28 terminals to a connector pane.

The most commonly used pattern is shown below. This pattern is used as a standard to assist in simplifying wiring.



Figure 4-4 shows an example of the standard layout used for terminal patterns. The top inputs and outputs are commonly used for passing references and the bottom inputs and outputs are used for error handling.

Figure 4-4. Example Terminal Pattern Layout



Note Avoid using connector panes with more than 16 terminals. Although connector pane patterns with more terminals might seem useful, they are very difficult to wire. If you need to pass more data, use clusters.

Assigning Terminals to Controls and Indicators

After you select a pattern to use for the connector pane, you can assign a front panel control or indicator to each of the connector pane terminals. When you assign controls and indicators to the connector pane, place inputs on the left and outputs on the right to prevent complicated or confusing wiring patterns.

To assign a terminal to a front panel control or indicator, click a terminal of the connector pane, then click the front panel control or indicator you want to assign to that terminal. Click an open space on the front panel window. The terminal changes to the data type color of the control to indicate that you connected the terminal.

You also can select the control or indicator first and then select the terminal.



Note Although you use the Wiring tool to assign terminals on the connector pane to front panel controls and indicators, no wires are drawn between the connector pane and these controls and indicators.

C. Using SubVIs

To place a subVI on the block diagram, click the **Select a VI** button on the **Functions** palette. Navigate to the VI you want to use as a subVI and double-click to place it on the block diagram.

You also can place an open VI on the block diagram of another open VI. Use the Positioning tool to click the icon in the upper right corner of the front panel or block diagram of the VI you want to use as a subVI and drag the icon to the block diagram of the other VI.

Opening and Editing SubVIs

To display the front panel of a subVI from the calling VI, use the Operating or Positioning tool to double-click the subVI on the block diagram. To display the block diagram of a subVI from the calling VI, press the <Ctrl> key and use the Operating or Positioning tool to double-click the subVI on the block diagram.

You can edit a subVI by using the Operating or Positioning tool to double-click the subVI on the block diagram. When you save the subVI, the changes affect all calls to the subVI, not just the current instance.

Setting Required, Recommended, and Optional Inputs and Outputs

In the **Context Help** window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button in the **Context Help** window.



You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals.

Right-click a terminal on the connector pane and select **This Connection Is** from the shortcut menu. A checkmark indicates the terminal setting. Select **Required**, **Recommended**, or **Optional**. You also can select **Tools»Options»Front Panel** and place a checkmark in the **Connector pane terminals default to required** checkbox. This option sets terminals on the connector pane to required instead of recommended. This applies to connections made using the wiring tool and to subVIs created using **Create SubVI**.



Note You can select **Dynamic Dispatch Input (Required)** or **Dynamic Dispatch Output (Recommended)** for dynamic dispatch member VIs.

For terminal inputs, required means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. Required is not available for terminal outputs. For terminal inputs and outputs, recommended or optional means that the block diagram on which you placed the subVI can execute if you do not wire the recommended or optional terminals. If you do not wire the terminals, the VI does not generate any warnings.

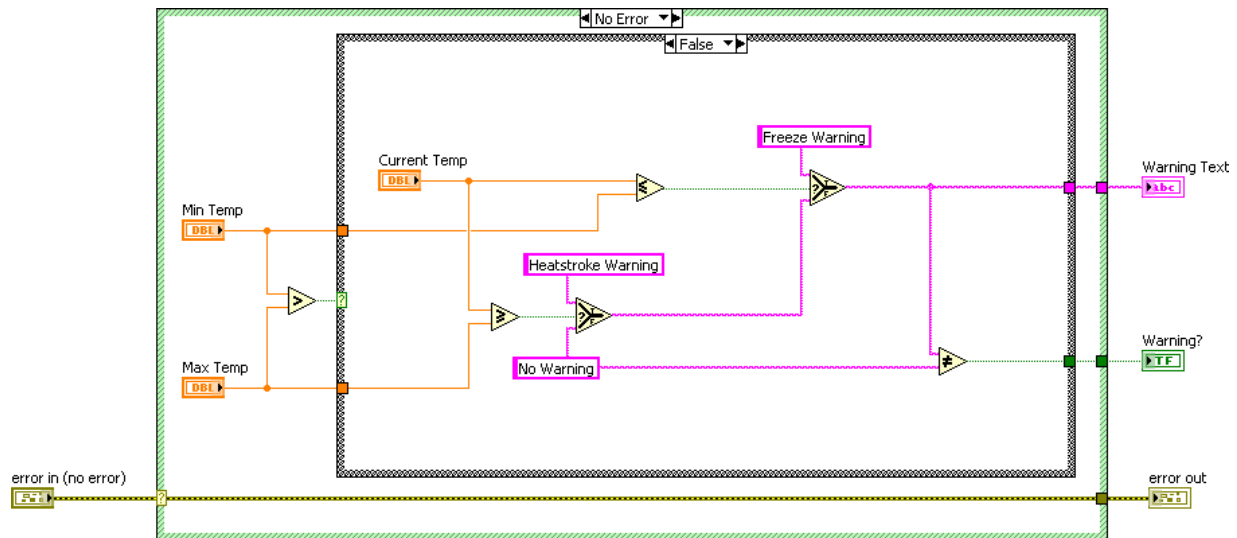
Inputs and outputs of VIs in `vi.lib` are already marked as **Required**, **Recommended**, or **Optional**. LabVIEW sets inputs and outputs of VIs you create to **Recommended** by default. Set a terminal setting to required only if the VI must have the input or output to run properly.

Handling Errors in SubVIs

You pass errors in and out of a subVI using error clusters. Using a Case structure, you handle errors passed into the subVI with a No Error case and Error case.

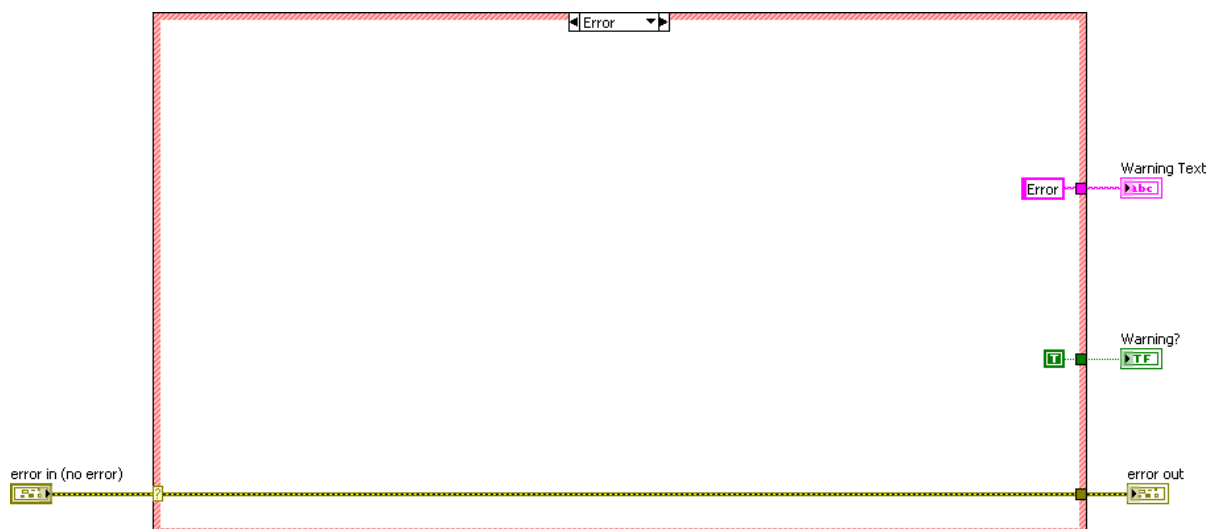
The No Error case, as shown in Figure 4-5, contains the code for the normal operation of the subVI.

Figure 4-5. No Error Case of Sample SubVI



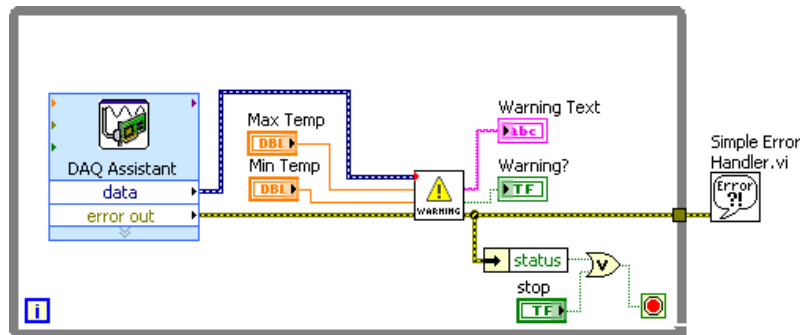
The Error case, as shown in Figure 4-6, typically passes the error from the **error in** cluster control to the **error out** cluster indicator and contains any code necessary to handle the error.

Figure 4-6. Error Case of Sample SubVI



Avoid using the Simple Error Handler VI and General Error Handler VI inside subVIs. If necessary, use these VIs in the calling VI, as shown in Figure 4-7.

Figure 4-7. Block Diagram of Calling VI

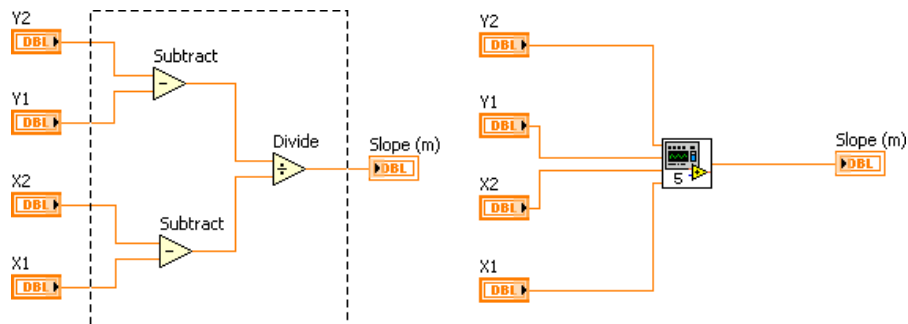


Creating a SubVI from an Existing VI

You can simplify the block diagram of a VI by converting sections of the block diagram into subVIs. Convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit>Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.

Figure 4-8 shows how to convert a selection into a subVI.

Figure 4-8. Creating a New SubVI



The new subVI uses a default pattern for the connector pane and a default icon. Double-click the subVI to edit the connector pane and icon, and to save the subVI.



Note Do not select more than 28 objects to create a subVI because 28 is the maximum number of connections on a connector pane. If your front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Self-Review: Quiz

1. On a subVI, which setting causes an error if the terminal is not wired?
 - a. Required
 - b. Recommended
 - c. Optional

2. You must create an icon to use a VI as a subVI.
 - a. True
 - b. False

Self-Review: Quiz Answers

1. On a subVI, which setting causes an error if the terminal is not wired?
 - a. **Required**
 - b. Recommended
 - c. Optional

2. You must create a custom icon to use a VI as a subVI.
 - a. True
 - b. **False**

You do not need to create a custom icon to use a VI as a subVI, but it is highly recommended to increase the readability of your code.

Notes

Creating and Leveraging Data Structures

Sometimes it is beneficial to group data related to one another. Use arrays and clusters to group related data in LabVIEW. Arrays combine data of the same data type into one data structure, and clusters combine data of multiple data types into one data structure. Use type definitions to define custom arrays and clusters. This lesson explains arrays, clusters, and type definitions, and applications where using these can be beneficial.

Topics

- A. Arrays
- B. Common Array Functions
- C. Polymorphism
- D. Auto-Indexing
- E. Clusters
- F. Type Definitions

A. Arrays

An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array. An array can have one or more dimensions and as many as $(2^{31}) - 1$ elements per dimension, memory permitting.

You can build arrays of numeric, Boolean, path, string, waveform, and cluster data types. Consider using arrays when you work with a collection of similar data and when you perform repetitive computations. Arrays are ideal for storing data you collect from waveforms or data generated in loops, where each iteration of a loop produces one element of the array.



Note Array indexes in LabVIEW are zero-based. The index of the first element in the array, regardless of its dimension, is zero.

Restrictions

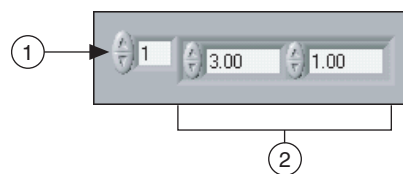
You cannot create arrays of arrays. However, you can use a multidimensional array or create an array of clusters where each cluster contains one or more arrays. Also, you cannot create an array of subpanel controls, tab controls, .NET controls, ActiveX controls, charts, or multi-plot XY graphs. Refer to the clusters section of this lesson for more information about clusters.

An example of a simple array is a text array that lists the twelve months of the year. LabVIEW represents this as a 1D array of strings with twelve elements.

Array elements are ordered. An array uses an index so you can readily access any particular element. The index is zero-based, which means it is in the range 0 to $n - 1$, where n is the number of elements in the array. For example, $n = 12$ for the twelve months of the year, so the index ranges from 0 to 11. March is the third month, so it has an index of 2.

Figure 5-1 shows an example of an array of numerics. The first element shown in the array (3.00) is at index 1, and the second element (1.00) is at index 2. The element at index 0 is not shown in this image, because element 1 is selected in the index display. The element selected in the index display always refers to the element shown in the upper left corner of the element display.

Figure 5-1. Array Control of Numerics



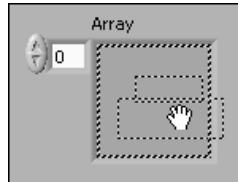
1 Index Display

2 Element Display

Creating Array Controls and Indicators

Create an array control or indicator on the front panel by adding an array shell to the front panel, as shown in the following front panel, and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, or cluster control or indicator, into the array shell.

Figure 5-2. Placing a Numeric Control in an Array Shell



If you attempt to drag an invalid control or indicator into the array shell, you are unable to place the control or indicator in the array shell.

You must insert an object in the array shell before you use the array on the block diagram. Otherwise, the array terminal appears black with an empty bracket and has no data type associated with it.

Creating Array Constants

To create an array constant on the block diagram, select an array constant on the **Functions** palette, place the array shell on the block diagram, and place a string constant, numeric constant, a Boolean constant, or cluster constant in the array shell. You can use an array constant to store constant data or as a basis for comparison with another array.

Two-Dimensional Arrays

A 2D array stores elements in a grid. It requires a column index and a row index to locate an element, both of which are zero-based. Figure 5-3 shows an 8 column by 8 row 2D array, which contains $8 \times 8 = 64$ elements.

Figure 5-3. 2D Array

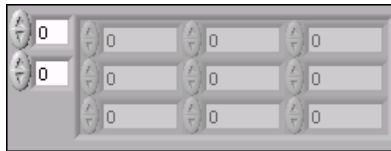
		Column Index							
		0	1	2	3	4	5	6	7
Row Index	0								
	1								
	2								
	3								
	4								
	5								
	6								
	7								

To add a multidimensional array to the front panel, right-click the index display and select **Add Dimension** from the shortcut menu. You also can resize the index display until you have as many dimensions as you want.

Initializing Arrays

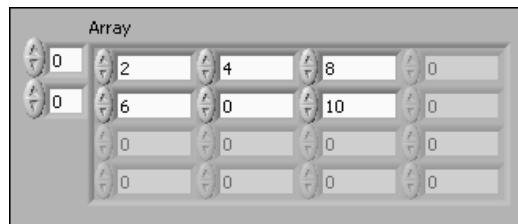
You can initialize an array or leave it uninitialized. When an array is initialized, you defined the number of elements in each dimension and the contents of each element. An uninitialized array contains a fixed number of dimensions but no elements. Figure 5-4 shows an uninitialized 2D array control. Notice that the elements are all dimmed. This indicates that the array is uninitialized.

Figure 5-4. 2D Uninitialized Array



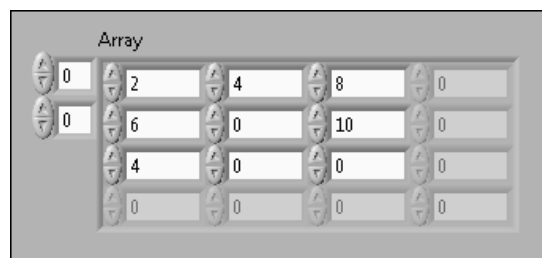
In Figure 5-5, six elements are initialized.

Figure 5-5. An Initialized 2D Array with Six Elements



In a 2D array, after you initialize an element in a row, the remaining elements in that row are initialized and populated with the default value for the data type. For example, in Figure 5-6, if you enter 4 into the element in the first column, third row, the elements in the second and third column in the third row are automatically populated with a 0.

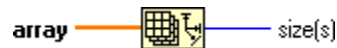
Figure 5-6. An Array Autopopulated with Zeroes



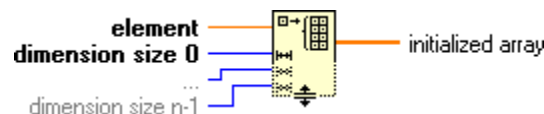
B. Common Array Functions

LabVIEW groups functions you can use to manipulate arrays on the **Array** palette. The following functions are the most common functions you use when you work with arrays.

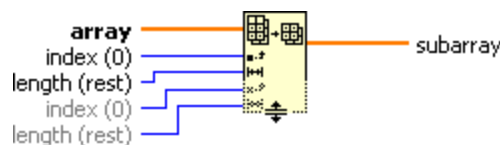
- **Array Size**—Returns the number of elements in each dimension of **array**. The connector pane displays the default data types for this polymorphic function.



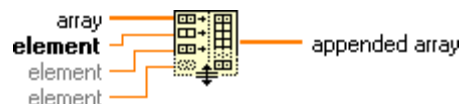
- **Initialize Array**—Creates an n-dimensional array in which every element is initialized to the value of **element**. Use the Positioning tool to resize the function and increase the number of dimensions (element, row, column, page, and so on) of the output array. The connector pane displays the default data types for this polymorphic function.



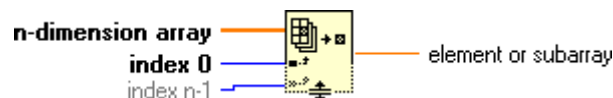
- **Array Subset**—Returns a portion of **array** starting at **index** and containing **length** elements.



- **Build Array**—Concatenates multiple arrays or appends elements to an n-dimensional array. You also can use the Replace Array Subset function to modify an existing array. The connector pane displays the default data types for this polymorphic function.



- **Index Array**—Returns the **element or subarray** of **n-dimension array** at **index**. When you wire an array to this function, the function resizes automatically to display **index** inputs for each dimension in the array you wire to **n-dimension array**. You also can add additional **element or subarray** terminals by resizing the function. The connector pane displays the default data types for this polymorphic function.



C. Polymorphism

Polymorphism is the ability of VIs and functions to automatically adapt to accept input data of different data types. Functions are polymorphic to varying degrees—none, some, or all of their inputs can be polymorphic. Some function inputs accept numeric values or Boolean values. Some accept numeric values or strings. Some accept not only scalar numeric values but also arrays of numeric values, clusters of numeric values, arrays of clusters of numeric values, and so on. Some accept only one-dimensional arrays although the array elements can be of any type. Some functions accept all types of data, including complex numeric values.

Polymorphism for Arithmetic Functions

The arithmetic functions take numeric input data. With some exceptions noted in the function descriptions, the default output has the same numeric representation as the input or, if the inputs have different representations, the default output is the larger of the inputs. For example, if you add an 8-bit integer and a 16-bit integer, the default output is a 16-bit integer. If you configure the output of a Numeric function, the settings you specify override the default behavior.

The arithmetic functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows:

Numeric type = numeric scalar OR array [*numeric type*] OR cluster [*numeric types*]

The numeric scalars can be floating-point numbers, integers, or complex floating-point numbers. LabVIEW does not allow you to use arrays of arrays.

Arrays can have any number of dimensions of any size. Clusters can have any number of elements. The output type of functions is of the same numeric representation as the input type. For functions with one input, the functions operate on each element of the array or cluster.

For functions with two inputs, you can use the following input combinations:

- **Similar**—Both inputs have the same structure, and the output has the same structure as the inputs.
- **One scalar**—One input is a numeric scalar, the other is an array or cluster, and the output is an array or cluster.
- **Array of**—One input is a numeric array, the other is the numeric type itself, and the output is an array.

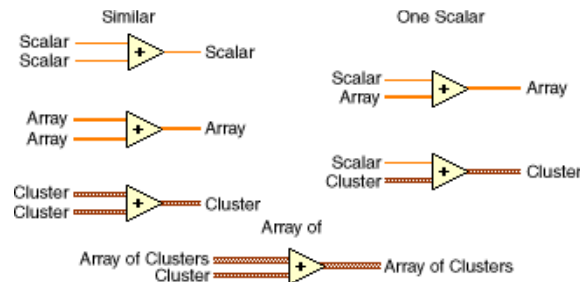
For similar inputs, LabVIEW performs the function on the respective elements of the structures. For example, LabVIEW can add two arrays element by element. Both arrays must have the same dimensionality. You can add arrays with differing numbers of elements; the output of such an addition has the same number of elements as the smallest input. Clusters must have the same number of elements, and the respective elements must be of the same type.

For operations involving a scalar and an array or cluster, LabVIEW performs the function on the scalar and the respective elements of the structure. For example, LabVIEW can subtract a number from all elements of an array, regardless of the dimensionality of the array.

For operations that involve a numeric type and an array of that type, LabVIEW performs the function on each array element. For example, a graph is an array of points, and a point is a cluster of two numeric types, x and y. To offset a graph by 5 units in the x direction and 8 units in the y direction, you can add a point, (5, 8), to the graph.

Figure 5-7 shows the possible polymorphic combinations of the Add function.

Figure 5-7. Polymorphic Combinations of the Add Function



Polymorphism for Boolean Functions

The logical functions accept Boolean input data, numeric input data, and error clusters. If the input is numeric, LabVIEW performs a bitwise operation. If the input is an integer, the output has the same representation. If the input is a floating-point number, LabVIEW rounds it to a 32-bit integer, and the output is a 32-bit integer. If the input is an error cluster, LabVIEW passes only the TRUE or FALSE value of the **status** parameter of the error cluster to the input terminal.

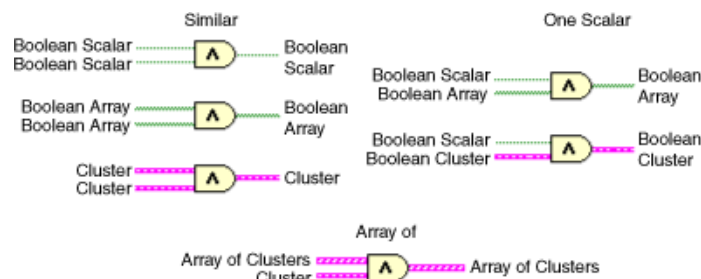
The logical functions work on arrays of numbers or Boolean values, clusters of numbers or Boolean values, arrays of clusters of numbers or Boolean values, and so on.

A formal and recursive definition of the allowable input type is as follows, except that complex numbers and arrays of arrays are not allowed:

$$\text{Logical type} = \text{Boolean scalar OR numeric scalar OR array [logical type] OR cluster [logical types]}$$

Logical functions with two inputs can have the same input combinations as the arithmetic functions. However, the logical functions have the further restriction that the base operations can only be between two Boolean values or two numbers. For example, you cannot have an AND between a Boolean value and a number. Figure 5-8 shows some combinations of Boolean values for the AND function.

Figure 5-8. Combinations of Boolean Values for the AND Function



D. Auto-Indexing

If you wire an array to or from a For Loop or While Loop, you can link each iteration of the loop to an element in that array by enabling auto-indexing. The tunnel image changes from a solid square to the image to indicate auto-indexing. Right-click the tunnel and select **Enable Indexing** or **Disable Indexing** from the shortcut menu to toggle the state of the tunnel.

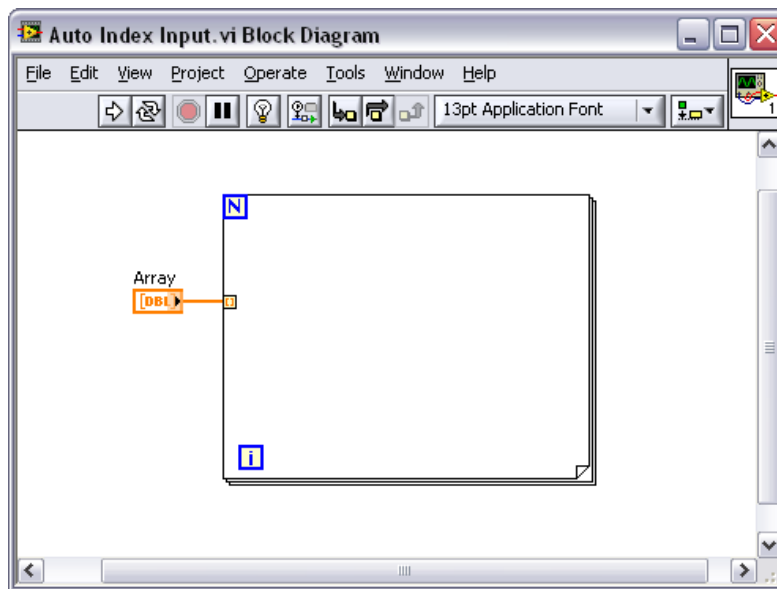


Array Inputs and Auto-Indexing

If you enable auto-indexing on an array wired to a For Loop input terminal, LabVIEW sets the count terminal to the array size so you do not need to wire the count terminal. Because you can use For Loops to process arrays one element at a time, LabVIEW enables auto-indexing by default for every array you wire to a For Loop. You can disable auto-indexing if you do not need to process arrays one element at a time.

In Figure 5-9, the For Loop executes a number of times equal to the number of elements in the array. Normally, if the count terminal of the For Loop is not wired, the run arrow is broken. However, in this case the run arrow is not broken.

Figure 5-9. Array Used to Set For Loop Count



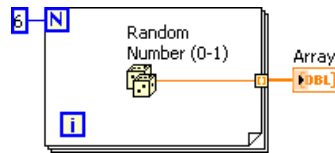
If you enable auto-indexing for more than one tunnel or if you wire the count terminal, the actual number of iterations becomes the smaller of the choices. For example, if two auto-indexed arrays enter the loop, with 10 and 20 elements respectively, and you wire a value of 15 to the count terminal, the loop still only executes 10 times, indexing all elements of the first array but only the first 10 elements of the second array.

Array Outputs and Auto-Indexing

When you auto-index an array output tunnel, the output array receives a new element from every iteration of the loop. Therefore, auto-indexed output arrays are always equal in size to the number of iterations.

The wire from the output tunnel to the array indicator becomes thicker as it changes to an array, as shown in Figure 5-10.

Figure 5-10. Auto-Indexed Output



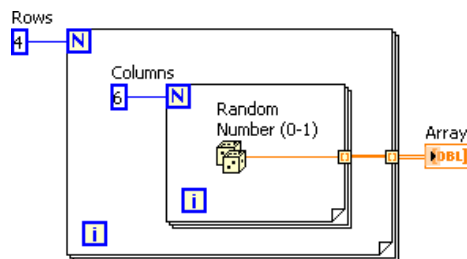
Right-click the tunnel at the loop border and select **Enable Indexing** or **Disable Indexing** from the shortcut menu to enable or disable auto-indexing. Auto-indexing for While Loops is disabled by default.

For example, disable auto-indexing if you need only the last value passed out of the tunnel.

Creating Two-Dimensional Arrays

You can use two For Loops, nested one inside the other, to create a 2D array. The outer For Loop creates the row elements, and the inner For Loop creates the column elements, as shown in Figure 5-11.

Figure 5-11. Creating a 2D Array

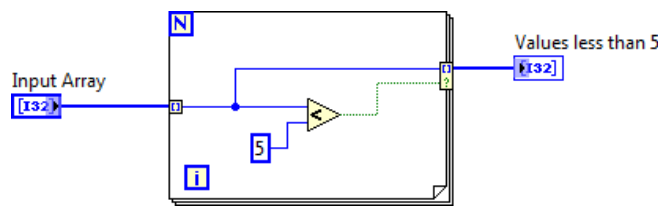


Auto-Indexing with a Conditional Tunnel

You can determine what values LabVIEW writes to the loop output tunnel based on a condition you specify by right-clicking the loop output tunnel and selecting **Tunnel Mode»Conditional** from the shortcut menu.

In Figure 5-12, the array **Input Array** contains the following elements: 7, 2, 0, 3, 1, 9, 5, and 7. Because of the conditional tunnel, the **Values less than 5** array contains only the elements 2, 0, 3, and 1 after this loop completes all iterations.

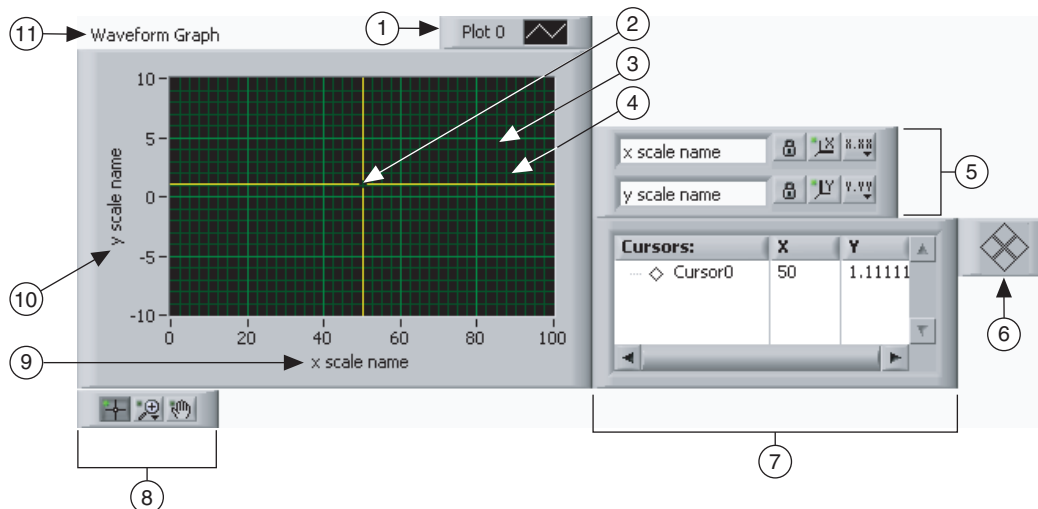
Figure 5-12. An Auto-Indexing For Loop with Conditional Tunnel



Waveform Graphs

VIs with a graph usually collect the data in an array and then plot the data to the graph. Figure 5-13 shows the elements of a graph.

Figure 5-13. Waveform Graph



1 Plot Legend	4 Mini-Grid Mark	7 Cursor Legend	10 Y-Scale
2 Cursor	5 Scale Legend	8 Graph Palette	11 Label
3 Grid Mark	6 Cursor Mover	9 X-Scale	

The graphs located on the Graph Indicators palette include the waveform graph and XY graph. The waveform graph plots only single-valued functions, as in $y = f(x)$, with points evenly distributed along the x-axis, such as acquired time-varying waveforms. XY graphs display any set of points, evenly sampled or not.

Resize the plot legend to display multiple plots. Use multiple plots to save space on the front panel and to make comparisons between plots. XY and waveform graphs automatically adapt to multiple plots.

Single Plot Waveform Graphs

The waveform graph accepts several data types for single-plot waveform graphs. The graph accepts a single array of values, interprets the data as points on the graph, and increments the x index by one starting at $x = 0$. The graph accepts a cluster of an initial x value, a delta x , and an array of y data. The graph also accepts the waveform data type, which carries the data, start time, and delta t of a waveform.

Refer to the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for examples of the data types that a waveform graph accepts.

Multi-plot Waveform Graphs

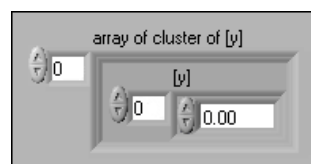
The waveform graph accepts several data types for displaying multiple plots. The waveform graph accepts a 2D array of values, where each row of the array is a single plot. The graph interprets the data as points on the graph and increments the x index by one, starting at $x = 0$. Wire a 2D array data type to the graph, right-click the graph, and select **Transpose Array** from the shortcut menu to handle each column of the array as a plot. This is particularly useful when you sample multiple channels from a DAQ device because the device can return the data as 2D arrays with each channel stored as a separate column.

Refer to the (Y) Multi Plot 1 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph also accepts a cluster of an initial x value, a delta x value, and a 2D array of y data. The graph interprets the y data as points on the graph and increments the x index by delta x , starting at the initial x value. This data type is useful for displaying multiple signals that are sampled at the same regular rate. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 2 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts a plot array where the array contains clusters. Each cluster contains a 1D array that contains the y data. The inner array describes the points in a plot, and the outer array has one cluster for each plot. The front panel in Figure 5-14 shows this array of the y cluster.

Figure 5-14. Array of the y Cluster



Use a plot array instead of a 2D array if the number of elements in each plot is different. For example, when you sample data from several channels using different time amounts from each channel, use this data structure instead of a 2D array because each row of a 2D array must have the same number of elements. The number of elements in the interior arrays of an array of clusters can vary. Refer to the (Y) Multi Plot 2 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts a cluster of an initial x value, a delta x value, and an array that contains clusters. Each cluster contains a 1D array that contains the y data. You use the Bundle function to bundle the arrays into clusters and you use the Build Array function to build the resulting clusters into an array. Refer to *Clusters* for more information about clusters and the Bundle function.

You also can use the Build Cluster Array function, which creates arrays of clusters that contain the inputs you specify. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 3 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph accepts an array of clusters of an x value, a delta x value, and an array of y data. This is the most general of the multiple-plot waveform graph data types because you can indicate a unique starting point and increment for the x -scale of each plot. Refer to the (Xo = 10, dX = 2, Y) Multi Plot 1 graph in the Waveform Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The waveform graph also accepts the dynamic data type, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes multiple channels, the graph displays a plot for each channel and automatically formats the plot legend and x -scale time stamp.

Single Plot XY Graphs

The XY graph accepts three data types for single-plot XY graphs. The XY graph accepts a cluster that contains an x array and a y array. Refer to the (X and Y arrays) Single Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The XY graph also accepts an array of points, where a point is a cluster that contains an x value and a y value. Refer to the (Array of Pts) Single Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type. The XY graph also accepts an array of complex data, in which the real part is plotted on the x -axis and the imaginary part is plotted on the y -axis.

Refer to *Clusters* for more information about clusters.

Multi-plot XY Graphs

The XY graph accepts three data types for displaying multiple plots. The XY graph accepts an array of plots, where a plot is a cluster that contains an x array and a y array. Refer to the (X and Y arrays) Multi Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

The XY graph also accepts an array of clusters of plots, where a plot is an array of points. A point is a cluster that contains an x value and a y value. Refer to the (Array of Pts) Multi Plot graph in the XY Graph VI in the `labview\examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type. The XY graph also accepts an array of clusters of plots, where a plot is an array of complex data, in which the real part is plotted on the x -axis and the imaginary part is plotted on the y -axis.

Refer to *Clusters* for more information about clusters.

E. Clusters

Clusters group data elements of mixed types. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value, and a string. A cluster is similar to a record or a struct in text-based programming languages.

Bundling several data elements into clusters eliminates wire clutter on the block diagram and reduces the number of connector pane terminals that subVIs need. The connector pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Most clusters on the block diagram have a pink wire pattern and data type terminal. Error clusters have a dark yellow wire pattern and data type terminal. Clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal. You can wire brown numeric clusters to Numeric functions, such as Add or Square Root, to perform the same operation simultaneously on all elements of the cluster.

Order of Cluster Elements

Although cluster and array elements are both ordered, you must unbundle all cluster elements at once using the Unbundle function. You can use the Unbundle By Name function to unbundle cluster elements by name. If you use the Unbundle By Name function, each cluster element must have a label. Clusters also differ from arrays in that they are a fixed size. Like an array, a cluster is either a control or an indicator. A cluster cannot contain a mixture of controls and indicators.

Creating Cluster Controls and Indicators

Create a cluster control or indicator on the front panel by adding a cluster shell to the front panel, as shown in the following front panel, and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, array, or cluster control or indicator, into the cluster shell.

Resize the cluster shell by dragging the cursor while you place the cluster shell.

Figure 5-15. Creation of a Cluster Control

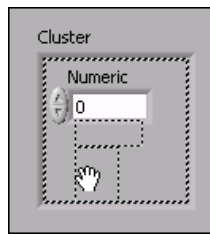
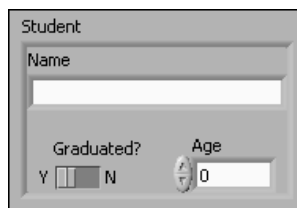


Figure 5-16 is an example of a cluster containing three controls: a string, a Boolean switch, and a numeric. A cluster is either a control or an indicator; it cannot contain a mixture of controls and indicators.

Figure 5-16. Cluster Control Example



Creating Cluster Constants

To create a cluster constant on the block diagram, select a cluster constant on the **Functions** palette, place the cluster shell on the block diagram, and place a string constant, numeric constant, a Boolean constant, or cluster constant in the cluster shell. You can use a cluster constant to store constant data or as a basis for comparison with another cluster.

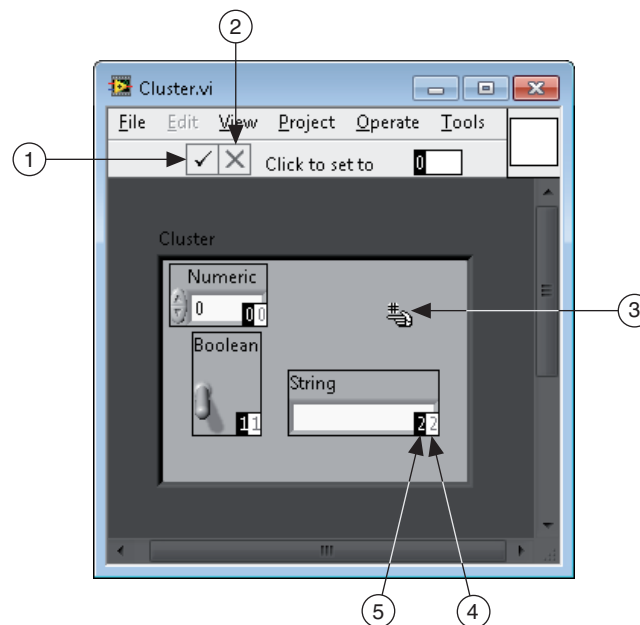
If you have a cluster control or indicator on the front panel window and you want to create a cluster constant containing the same elements on the block diagram, you can either drag that cluster from the front panel window to the block diagram or right-click the cluster on the block diagram and select **Create»Constant** from the shortcut menu.

Cluster Order

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions on the block diagram. You can view and modify the cluster order by right-clicking the cluster border and selecting **Reorder Controls In Cluster** from the shortcut menu.

The toolbar and cluster change, as shown in Figure 5-17.

Figure 5-17. Reordering a Cluster



1 Confirm Button	3 Cluster Order Cursor	5 New Order
2 Cancel Button	4 Current Order	

The white box on each element shows its current place in the cluster order. The black box shows the new place in the order for an element. To set the order of a cluster element, enter the new order number in the **Click to set to** text box and click the element. The cluster order of the element changes, and the cluster order of other elements adjusts. Save the changes by clicking the **Confirm** button on the toolbar. Revert to the original order by clicking the **Cancel** button.

Autosizing Clusters

Complete the following steps to shrink clusters to fit their contents.

1. Right-click a cluster shell border and select **AutoSizing»Size to Fit** from the shortcut menu.
2. Add another element to the cluster shell. When **AutoSizing»Size to Fit** is selected, the cluster resizes itself if necessary when you add another element to the cluster shell.
3. Right-click the cluster shell border and select **AutoSizing»Arrange Horizontally** or **AutoSizing»Arrange Vertically** from the shortcut menu to arrange the elements in the cluster horizontally or vertically.

Using Cluster Functions

Use the Cluster functions to create and manipulate clusters. For example, you can perform tasks similar to the following:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

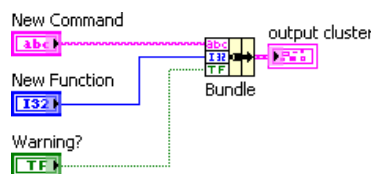
Use the Bundle function to assemble a cluster, use the Bundle function and Bundle By Name function to modify a cluster, and use the Unbundle function and the Unbundle By Name function to disassemble clusters.

You also can place the Bundle, Bundle By Name, Unbundle, and Unbundle By Name functions on the block diagram by right-clicking a cluster terminal on the block diagram and selecting **Cluster, Class & Variant Palette** from the shortcut menu. The Bundle and Unbundle functions automatically contain the correct number of terminals. The Bundle By Name and Unbundle By Name functions appear with the first element in the cluster. Use the Positioning tool to resize the Bundle By Name and Unbundle By Name functions to show the other elements of the cluster.

Assembling Clusters

Use the Bundle function to assemble a cluster from individual elements or to change the values of individual elements in an existing cluster without having to specify new values for all elements. Use the Positioning tool to resize the function or right-click an element input and select **Add Input** from the shortcut menu.

Figure 5-18. Assembling a Cluster on the Block Diagram



Disassembling Clusters

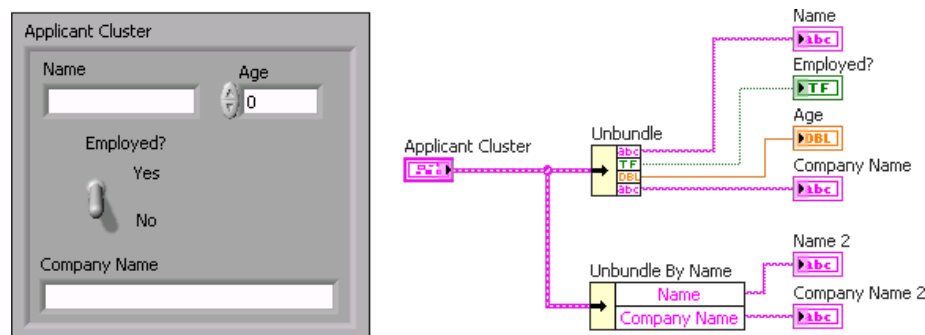
Use the Unbundle function to split a cluster into its individual elements.

Use the Unbundle By Name function to return the cluster elements whose names you specify. The number of output terminals does not depend on the number of elements in the input cluster.

While the Unbundle function displays all parts of a cluster, the Unbundle by Name function can be set to display specific elements of a cluster only. To select which elements the Unbundle by Name function should display, use the Operating tool to click an output terminal of the function and select an element from the pull-down menu. You also can right-click the output terminal and select the element from the **Select Item** shortcut menu.

For example, if you use the Unbundle function with the cluster in Figure 5-19, it has four output terminals that correspond to the four controls in the cluster. You must know the cluster order so you can associate the correct Boolean terminal of the unbundled cluster with the corresponding switch in the cluster. In this example, the elements are ordered from top to bottom starting with element 0. If you use the Unbundle By Name function, you can have an arbitrary number of output terminals and access individual elements by name in any order.

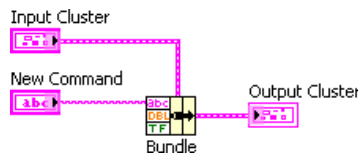
Figure 5-19. Unbundle and Unbundle By Name



Modifying a Cluster

If you wire the cluster input, you can wire only the elements you want to change. For example, the Input Cluster shown in Figure 5-20 contains three controls.

Figure 5-20. Bundle Used to Modify a Cluster



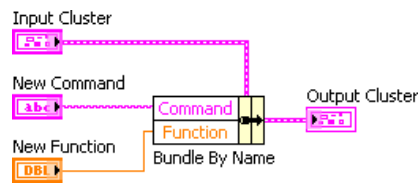
If you know the cluster order, you can use the Bundle function to change the **Command** value by wiring the elements shown in Figure 5-20.

You can also use the Bundle By Name function to replace or access labeled elements of an existing cluster. The Bundle By Name function works like the Bundle function, but instead of referencing cluster elements by their cluster order, it references them by their owned labels. You can access only elements with owned labels. The number of inputs does not need to match the number of elements in **output cluster**.

While the Bundle function displays all parts of a cluster, the Bundle by Name function can be set to display specific elements of a cluster only. To select which elements the Bundle by Name function should display, use the Operating tool to click an output terminal of the function and select an element from the pull-down menu. You also can right-click the output terminal and select the element from the **Select Item** shortcut menu.

In Figure 5-21, you can use the Bundle By Name function to update the values of **Command** and **Function** with the values of **New Command** and **New Function**.

Figure 5-21. Bundle By Name Used to Modify a Cluster



Use the Bundle By Name function for data structures that might change during development. If you add a new element to the cluster or modify its order, you do not need to rewire the Bundle By Name function because the names are still valid.

Error Clusters

LabVIEW contains a custom cluster called the error cluster. LabVIEW uses error clusters to pass error information.

For more information about using error clusters, refer to Lesson 2, *Troubleshooting and Debugging VIs*, of this manual and the *Handling Errors* topic of the *LabVIEW Help*.

F. Type Definitions

You can use type definitions to define custom arrays and clusters. A type definition is a master copy of a custom data type (custom control, indicator, or constant) that multiple VIs can use. When you use type definitions, you can update all instances or copies of the custom data type in a single edit.

Custom Controls and Indicators

Use custom controls and indicators to extend the available set of front panel objects. You can create custom controls and indicators for an application that vary cosmetically from built-in LabVIEW controls and indicators. You can save a custom control or indicator you create in a directory or LLB and use the custom control or indicator on other front panels. You also can create an icon for the custom control or indicator and add it to the **Controls** palette.

Refer to the *Creating Custom Controls, Indicators, and Type Definitions* topic of the *LabVIEW Help* for more information about creating and using custom controls and type definitions.

Use the Control Editor window to customize controls and indicators. For example, you can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.

You can display the Control Editor window in the following ways:

- Right-click a front panel control or indicator and select **Advanced»Customize** from the shortcut menu.
- Use the Positioning tool to select a front panel control or indicator and select **Edit»Customize Control**.
- Use the **New** dialog box.

The Control Editor appears with the selected front panel object in its window. The Control Editor has two modes, edit mode and customize mode.

The Control Editor window toolbar indicates whether you are in edit mode or in customize mode. The Control Editor window opens in edit mode. Click the **Change to Customize Mode** button to change to customize mode. To change back to edit mode, click the **Change to Edit Mode** button. You also can switch between modes by selecting **Operate»Change to Customize Mode** or **Operate»Change to Edit Mode**.

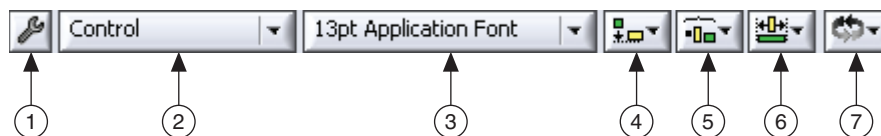


Use edit mode to change the size or color of a control or indicator and to select options from its shortcut menu, just as you do in edit mode on a front panel.

Use customize mode to make extensive changes to controls or indicators by changing the individual parts of a control or indicator.

Edit Mode

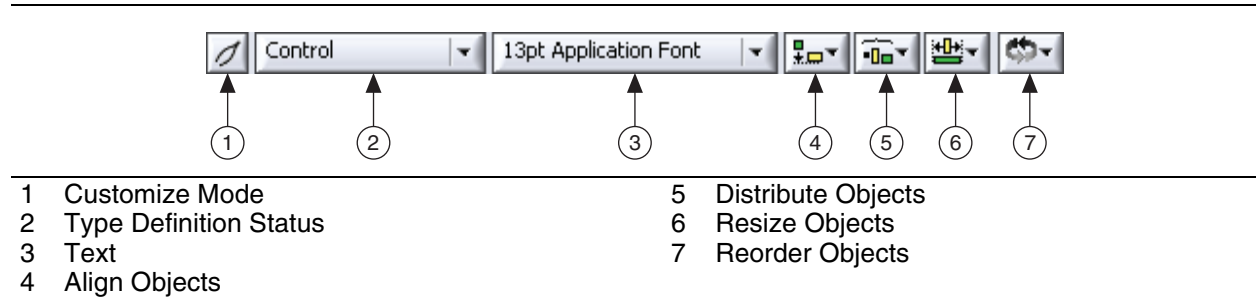
In the edit mode, you can right-click the control and manipulate its settings as you would in the LabVIEW programming environment.



1	Edit Mode	5	Distribute Objects
2	Type Definition Status	6	Resize Objects
3	Text	7	Reorder Objects
4	Align Objects		

Customize Mode

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



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

Saving Custom Controls

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

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

Type Definitions

Use type definitions and strict type definitions to link all the instances of a custom control or indicator to a saved custom control or indicator file. You can make changes to all instances of the custom control or indicator by editing only the saved custom control or indicator file, which is useful if you use the same custom control or indicator in several VIs.

When you place a custom control or indicator in a VI, no connection exists between the custom control or indicator you saved and the instance of the custom control or indicator in the VI. Each instance of a custom control or indicator is a separate, independent copy. Therefore, changes you make to a custom control or indicator file do not affect VIs already using that custom control or indicator. If you want to link instances of a custom control or indicator to the custom control or indicator file, save the custom control or indicator as a type definition or strict type definition. All instances of a type definition or a strict type definition link to the original file from which you created them.

When you save a custom control or indicator as a type definition or strict type definition, any data type changes you make to the type definition or strict type definition affect all instances of the type

definition or strict type definition in all the VIs that use it. Also, cosmetic changes you make to a strict type definition affect all instances of the strict type definition on the front panel.

Type definitions identify the correct data type for each instance of a custom control or indicator. When the data type of a type definition changes, all instances of the type definition automatically update. In other words, the data type of the instances of the type definition change in each VI where the type definition is used. However, because type definitions identify only the data type, only the values that are part of the data type update. For example, on numeric controls, the data range is not part of the data type. Therefore, type definitions for numeric controls do not define the data range for the instances of the type definitions. Also, because the item names in ring controls do not define the data type, changes to ring control item names in a type definition do not change the item names in instances of the type definition. However, if you change the item names in the type definition for an enumerated type control, the instances update because the item names are part of the data type. An instance of a type definition can have its own unique caption, label, description, tip strip, default value, size, color, or style of control or indicator, such as a knob instead of a slide.

If you change the data type in a type definition, LabVIEW converts the old default value in instances of the type definition to the new data type, if possible. LabVIEW cannot preserve the instance default value if the data type changes to an incompatible type, such as replacing a numeric control or indicator with a string control or indicator. When the data type of a type definition changes to a data type incompatible with the previous type definition, LabVIEW sets the default value of instances to the default value you specify in the `.ctl` file. If you do not specify a default value, LabVIEW uses the default value for the data type. For example, if you change a type definition from a numeric to a string type, LabVIEW replaces any default values associated with the old numeric data type with empty strings.

Strict Type Definitions

A strict type definition forces everything about an instance to be identical to the strict type definition, except the caption, label, description, tip strip, and default value. As with type definitions, the data type of a strict type definition remains the same everywhere you use the strict type definition. Strict type definitions also define other values, such as range checking on numeric controls and the item names in ring controls. The only VI Server properties available for strict type definitions are those that affect the appearance of the control or indicator, such as Visible, Disabled, Key Focus, Blinking, Position, and Bounds.

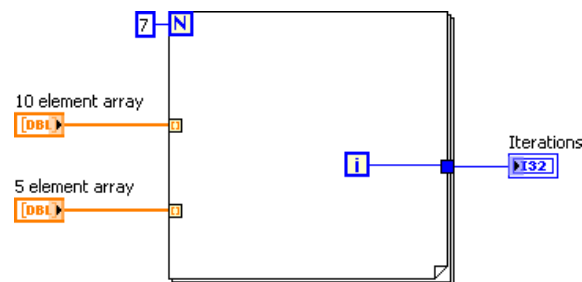
You cannot prevent an instance of a strict type definition from automatically updating unless you remove the link between the instance and the strict type definition.

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

Self-Review: Quiz

1. You can create an array of arrays.
 - a. True
 - b. False
2. You have two input arrays wired to a For Loop. Auto-indexing is enabled on both tunnels. One array has 10 elements, the second array has five elements. A value of 7 is wired to the Count terminal, as shown in Figure 5-22. What is the value of the **Iterations** indicator after running this VI?

Figure 5-22. What is the Value of the Iterations Indicator?



3. Which of the following custom control settings defines the data type of all instances of a control but allows for different colors and font styles?
 - a. Control
 - b. Type Definition
 - c. Strict Type Definition
 - d. Cluster control
4. You have input data representing a circle: X Position, Y Position, and Radius. In the future, you might need to modify your data to include the color of the circle.
 What data structure should you use to represent the circle in your application?
 - a. Three separate controls for the two positions and the radius.
 - b. A cluster containing all of the data.
 - c. A custom control containing a cluster.
 - d. A type definition containing a cluster.
 - e. An array with three elements.

Self-Review: Quiz Answers

1. You can create an array of arrays.

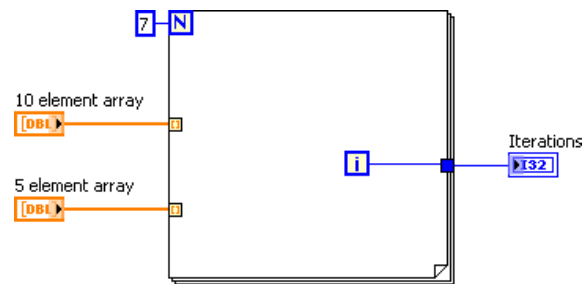
a. True

b. False

You cannot drag an array data type into an array shell. However, you can create two-dimensional arrays.

2. You have two input arrays wired to a For Loop. Auto-indexing is enabled on both tunnels. One array has 10 elements, the second array has five elements. A value of 7 is wired to the Count terminal, as shown in Figure 5-23. What is the value of the **Iterations** indicator after running this VI?

Figure 5-23. What is the value of the Iterations indicator?



Value of Iterations = 4

LabVIEW does not exceed the array size. This helps to protect against programming error. LabVIEW mathematical functions work the same way—if you wire a 10 element array to the **x** input of the Add function, and a 5 element array to the **y** input of the Add function, the output is a 5 element array.

Although the for loop runs 5 times, the iterations are zero based, therefore the value of the Iterations indicators is 4.

3. Which of the following custom control settings defines the data type of all instances of a control but allows for different colors and font styles?

a. Control

b. Type Definition

c. Strict Type Definition

d. Cluster control

4. You have input data representing a circle: X Position, Y Position, and Radius. In the future, you might need to modify your data to include the color of the circle.

What data structure should you use to represent the circle in your application?

- a. Three separate controls for the two positions and the radius.
- b. A cluster containing all of the data.
- c. A custom control containing a cluster.
- d. A type definition containing a cluster.**
- e. An array with three elements.

Notes

Notes

Managing File and Hardware Resources

You have learned how to acquire data and how to display it, but storage of your data is usually a very important part of any project. You have also learned how to set up your hardware and configure it in the Measurement & Automation Explorer. In this lesson, you learn about storing data, programming a basic DAQ application using the DAQmx API, and controlling stand-alone instruments with the VISA API and instrument drivers in LabVIEW.

Topics

- A. Understanding Hardware and Software Resources
- B. File I/O
- C. Acquiring Measurements with a DAQ System
- D. Controlling Instruments

A. Understanding Hardware and Software Resources

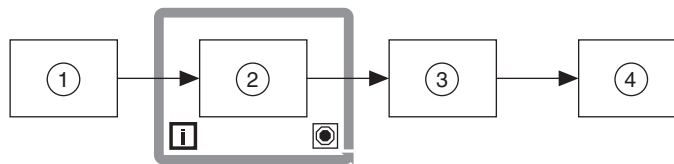
A resource is an addressable file, hardware device, object, or network connection available on the system. The system identifies resources using paths, names, ports or other identifiers. Using these identifiers and the VIs and functions that LabVIEW provides, you can program your system to access those resources.

Accessing Resources in LabVIEW

A typical resource operation involves the following operations as shown in Figure 6-1:

1. Creating a resource.
2. Reading and writing to a resource.
3. Closing the resource.
4. Checking for errors from the resource.

Figure 6-1. typical Resource Operation



- 1 Open, Initialize or create resource—LabVIEW creates a reference number (refnum) as a unique identifier for the resource
- 2 Read/Write to resource
- 3 Close resource—The refnum becomes obsolete
- 4 Check for errors—Display any errors from the resource



Tip The function or VI names associated with creating a resource generally include one of the following verbs: open, initialize, or create. The function or VI that creates the resource takes a resource path or device name as an input and then creates a unique identifier to the resource.

Reference Numbers (Refnums)

A reference number, or refnum, is a unique identifier for a resource. When you open a file, device, or network connection, LabVIEW creates a refnum associated with that file, device, or network connection. All operations you perform on open files, devices, or network connections use the refnums to identify each object. Use a refnum control to pass a refnum into or out of a VI. For example, use a refnum control to modify the contents of the file that a refnum is referencing without closing and reopening the file.

Because a refnum is a temporary pointer to an open resource, it is valid only for the period during which the resource is open. If you close the resource, LabVIEW disassociates the refnum from the resource, and the refnum becomes obsolete. If you open the resource again, LabVIEW creates a

new refnum that is different from the first refnum. LabVIEW allocates memory for a resource that is associated with a refnum. Close the refnum to release the resource from memory.

LabVIEW remembers information associated with each refnum, such as the current location for reading from or writing to the object and the degree of user access, so you can perform concurrent but independent operations on a single resource. If a VI opens an object multiple times, each open operation returns a different refnum. LabVIEW automatically closes refnums for you when a VI finishes running, but it is a good programming practice to close refnums as soon as you are finished with them to most efficiently use memory and other resources. Close refnums in the opposite order that you opened them. For example, if you obtain a refnum to object A and invoke a method on object A to obtain a refnum to object B, close the refnum to object B first and then close the refnum to object A.

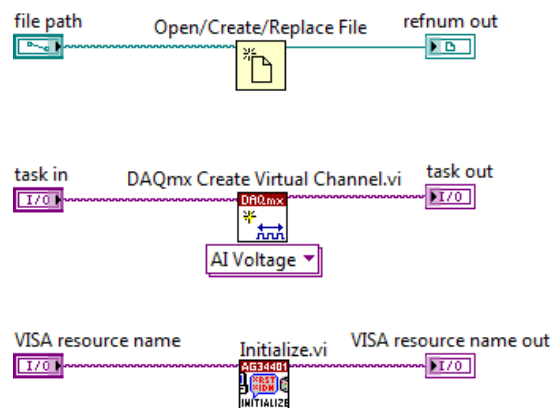
Examples of Refnum APIs

When interfacing with a DAQ device using the DAQmx driver API, the reference is referred to as a task. A task is a collection of one or more virtual channels with timing, triggering, and other properties. Conceptually, a task represents a measurement or generation you want to perform.

When controlling an instrument using the VISA API, the reference is referred to as a VISA session. The VISA session is maintained by the VISA resource name control.

Figure 6-2 illustrates several different APIs and the refnums they use to access resources.

Figure 6-2. Example Refnum Wiring



B. File I/O

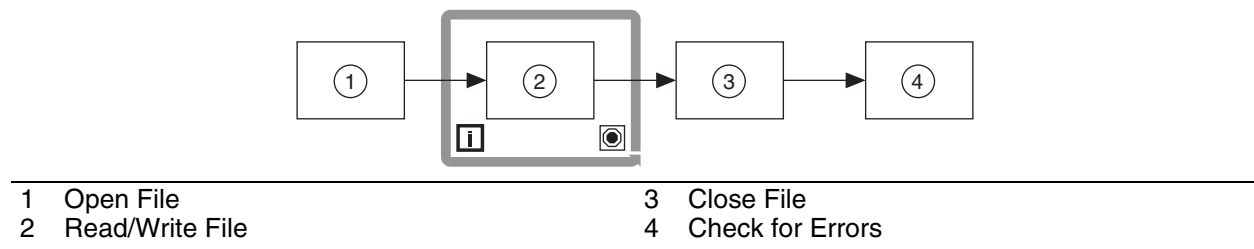
File I/O operations pass data to and from files. Use the File I/O VIs and functions to handle all aspects of file I/O.

Understanding File I/O

A typical file I/O operation involves the following process as shown in Figure 6-3.

1. You create or open a file. After the file opens, a unique identifier called a refnum represents the file.
2. The File I/O VI or function reads from or writes to the file.
3. You close the file.

Figure 6-3. Steps in a Typical File I/O Operation



File Formats

LabVIEW can use or create the following file formats: Binary, ASCII, LVM, and TDMS.

- **Binary**—Binary files are the underlying file format of all other file formats.
- **ASCII**—An ASCII file is a specific type of binary file that is a standard used by most programs. It consists of a series of ASCII codes. ASCII files are also called text files.
- **LVM**—The LabVIEW measurement data file (.lvm) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The .lvm file includes information about the data, such as the date and time the data was generated. This file format is a specific type of ASCII file created for LabVIEW.
- **TDMS**—This file format is a specific type of binary file created for National Instruments products. It actually consists of two separate files—a binary file that contains data and stores properties about the data, and a binary index file that provides consolidated information on all the attributes and pointers in the binary file.

In this course, you learn about creating text (ASCII) files. Use text files when you want to access the file from another application, if disk space and file I/O speed are not crucial, if you do not need to perform random access read or writes, and if numeric precision is not important.

You used an LVM file in Lesson 1, *Navigating LabVIEW*. To learn more about binary and TDMS files, refer to the *LabVIEW Help* or the *LabVIEW Core 2* course.

Understanding High-Level File I/O

Some File I/O VIs perform all three steps of a file I/O process—open, read/write, and close. If a VI performs all three steps, it is referred to as a high-level VI. However, these VIs may not be as efficient as the low-level VIs and functions designed for individual parts of the process. If you are writing to a file in a loop, use low-level file I/O VIs. If you are writing to a file in a single operation, you can use the high-level file I/O VIs instead.

LabVIEW includes the following high-level file I/O VIs:

- **Write to Spreadsheet File**—Converts a 2D or 1D array of double-precision numbers to a text string and writes the string to a new ASCII file or appends the string to an existing file. You also can transpose the data. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to create a text file readable by most spreadsheet applications.
- **Read From Spreadsheet File**—Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D double-precision array of numbers. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format.
- **Write to Measurement File**—An Express VI that writes data to a text-based measurement file (.lvm) or a binary measurement file (.tdms) format. You can specify the save method, file format (.lvm or .tdms), header type, and delimiter.
- **Read from Measurement File**—An Express VI that reads data from a text-based measurement file (.lvm) or a binary measurement file (.tdms) format. You can specify the file name, file format, and segment size.



Tip Avoid placing the high-level VIs in loops, because the VIs perform open and close operations each time they run.

Understanding Low-Level File I/O

Low-level file I/O VIs and functions each perform only one piece of the file I/O process. For example, there is one function to open an ASCII file, one function to read an ASCII file, and one function to close an ASCII file. Use low-level functions when file I/O is occurring within a loop.

Disk Streaming with Low-Level Functions

You also can use File I/O functions for disk streaming operations, which save memory resources by reducing the number of times the function interacts with the operating system to open and close the file. Disk streaming is a technique for keeping files open while you perform multiple write operations, for example, within a loop.

Wiring a path control or a constant to the Write to Text File function, the Write to Binary File function, or the Write To Spreadsheet File VI adds the overhead of opening and closing the file each time the function or VI executes. VIs can be more efficient if you avoid opening and closing the same files frequently.

To avoid opening and closing the same file, you need to pass a refnum to the file into the loop. When you open a file, device, or network connection, LabVIEW creates a refnum associated with that file, device, or network connection. All operations you perform on open files, devices, or network connections use the refnums to identify each object.

The examples in Figure 6-4 and Figure 6-5 show the advantages of using disk streaming. In Figure 6-4, the VI must open and close the file during each iteration of the loop. Figure 6-5 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.

Figure 6-4. Non-Disk Streaming Example

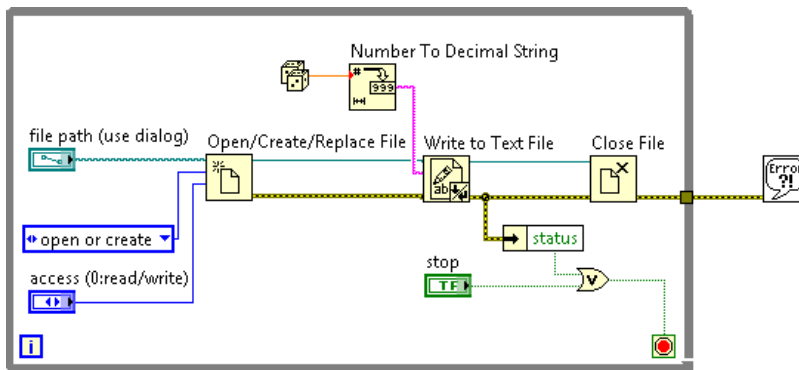
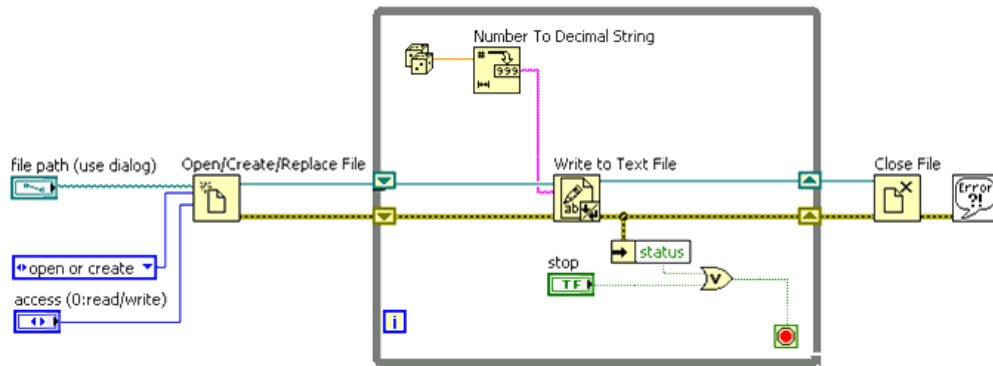


Figure 6-5. Disk Streaming Example



LabVIEW Data Directory

You can use the default LabVIEW Data directory to store the data files LabVIEW generates, such as .lvn or .txt files. LabVIEW installs the LabVIEW Data directory in the default file directory for your operating system to help you organize and locate the data files LabVIEW generates. By default, the Write LabVIEW Measurement File Express VI stores the .lvn files it generates in this directory, and the Read LabVIEW Measurement File Express VI reads from this directory. The Default Data Directory constant and the Application:Default:Data Directory property also return the LabVIEW Data directory by default.



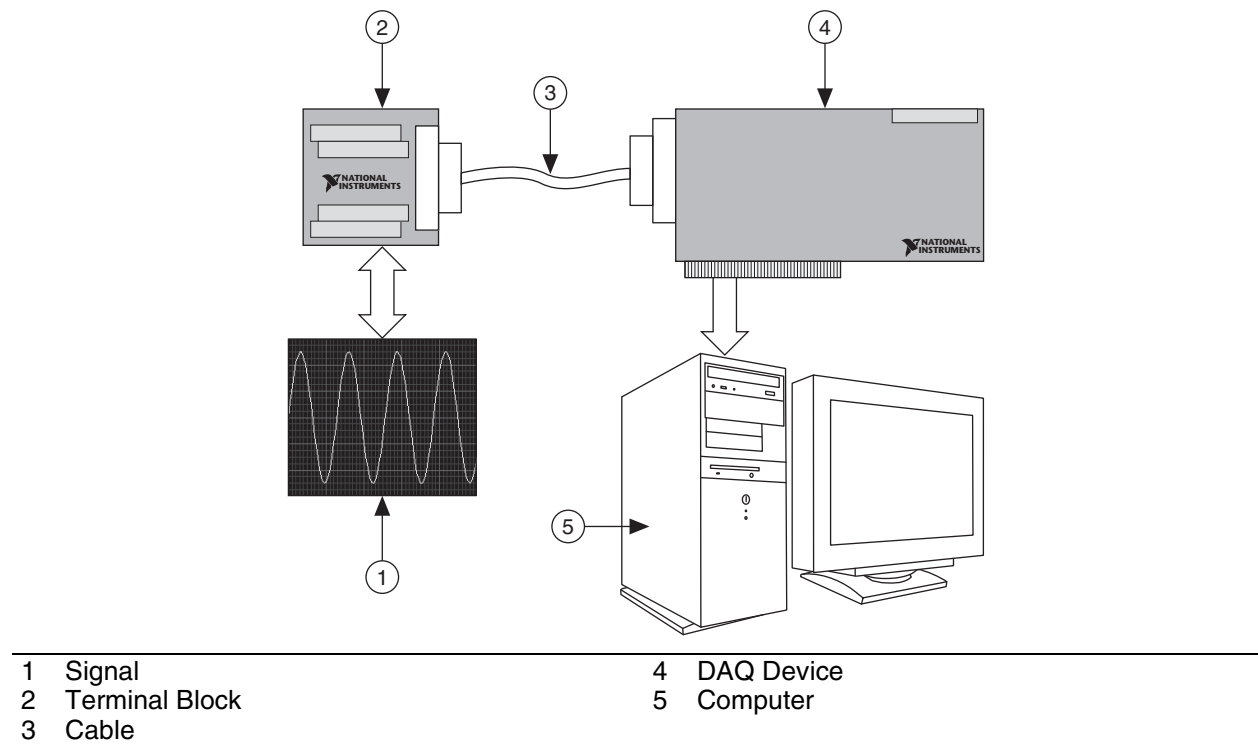
Select **Tools»Options** and select **Paths** from the **Category** list to specify a different default data directory. The default data directory differs from the default directory, which is the directory you specify for new VIs, custom controls, VI templates, or other LabVIEW documents you create.

C. Acquiring Measurements with a DAQ System

A data acquisition (DAQ) system uses a data acquisition device to pass a conditioned electrical signal to a computer for software analysis and data logging. You can choose a data acquisition device that uses a PCI bus, a PCI Express bus, a PXI bus, or the computer USB or IEEE 1394 port. This section explains the hardware used in a data acquisition system and how to configure the devices.

A typical DAQ system has three basic types of hardware—a terminal block, a cable, and a DAQ device, as shown in Figure 6-6.

Figure 6-6. Typical DAQ System



After you have converted a physical phenomenon into a measurable signal with or without signal conditioning, you need to acquire that signal. To acquire a signal, you need a terminal block, a cable, a DAQ device, and a computer. This hardware combination can transform a standard computer into a measurement and automation system.

Using a Terminal Block and Cable

A terminal block provides a place to connect signals. It consists of screw or spring terminals for connecting signals and a connector for attaching a cable to connect the terminal block to a DAQ device. Terminal blocks have 100, 68, or 50 terminals. The type of terminal block you should choose depends on two factors—the device and the number of signals you are measuring.

A terminal block with 68 terminals offers more ground terminals to connect a signal to than a terminal block with 50 terminals. Having more ground terminals prevents the need to overlap wires to reach a ground terminal, which can cause interference between the signals.

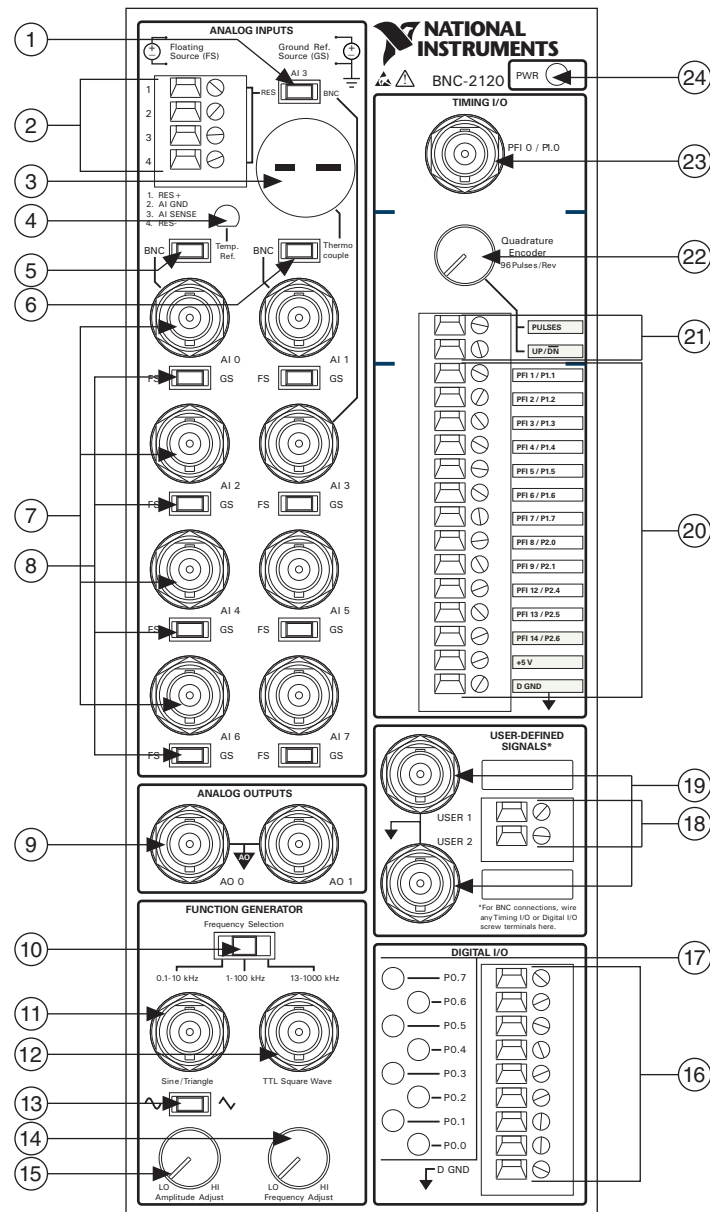
Terminal blocks can be shielded or non-shielded. Shielded terminal blocks offer better protection against noise. Some terminal blocks contain extra features, such as cold-junction compensation, that are necessary to properly measure a thermocouple.

A cable transports the signal from the terminal block to the DAQ device. Cables come in 100-, 68-, and 50-pin configurations. Choose a configuration depending on the terminal block and the DAQ device you are using. Cables, like terminal blocks, are shielded or non-shielded.

Refer to the DAQ section of the National Instruments catalog or to ni.com/products for more information about specific types of terminal blocks and cables.

BNC-2120

Figure 6-7 shows the BNC-2120, which can be used to complete the exercises in this course.

Figure 6-7. BNC-2120

- | | |
|--|---------------------------------------|
| 1 RES/BNC Switch (AI 3) | 13 Sine/Triangle Waveform Switch |
| 2 Resistor Measurement Screw Terminals | 14 Frequency Adjust Knob |
| 3 Thermocouple Input Connector | 15 Amplitude Adjust Knob |
| 4 Temperature Reference | 16 Digital I/O Screw Terminals |
| 5 BNC/Temp. Ref. Switch (AI 0) | 17 Digital I/O LEDs |
| 6 BNC/Thermocouple Switch (AI 1) | 18 User-Defined Screw Terminals |
| 7 Analog Input BNC Connectors | 19 User-Defined BNC Connectors |
| 8 FS/GS Switches | 20 Timing I/O Screw Terminals |
| 9 Analog Output BNC Connector | 21 Quadrature Encoder Screw Terminals |
| 10 Frequency Range Selection Switch | 22 Quadrature Encoder Knob |
| 11 Sine/Triangle BNC Connector | 23 Timing I/O BNC Connector |
| 12 TTL Square Wave BNC Connector | 24 Power Indicator LED |

Using DAQ Devices

Most DAQ devices have four standard elements—analog input, analog output, digital I/O, and counters.

You can transfer the signal you measure with the DAQ device to the computer through a variety of different bus structures. For example, you can use a DAQ device that plugs into the PCI or PCI Express bus of a computer, a DAQ device connected to the PCMCIA socket of a laptop, or a DAQ device connected to the USB port of a computer. You also can use PXI/CompactPCI to create a portable, versatile, and rugged measurement system.

If you do not have a DAQ device, you can simulate one in Measurement & Automation Explorer (MAX) to complete your software testing. You learn to simulate a device in the *Simulating a DAQ Device* section of this lesson.

Refer to the DAQ section of the NI catalog or to ni.com/products for more information about specific types of DAQ devices.

Analog Input

Analog input is the process of measuring an analog signal and transferring the measurement to a computer for analysis, display or storage. An analog signal is a signal that varies continuously. Analog input is most commonly used to measure voltage or current. You can use many types of devices to perform analog input, such as multifunction DAQ (MIO) devices, high-speed digitizers, digital multimeters (DMMs) and Dynamic Signal Acquisition (DSA) devices.

Acquiring an analog signal with a computer requires a process known as analog-to-digital conversion, which takes an electrical signal and translates it into digital data so that a computer can process it. Analog-to-digital converters (ADCs) are circuit components that convert a voltage level into a series of ones and zeroes.

ADCs sample the analog signal on each rising or falling edge of a sample clock. In each cycle, the ADC takes a snapshot of the analog signal, so that the signal can be measured and converted into a digital value. A sample clock controls the rate at which samples of the input signal are taken. Because the incoming, or unknown signal is a real world signal with infinite precision, the ADC approximates the signal with fixed precision. After the ADC obtains this approximation, the approximation can be converted to a series of digital values. Some conversion methods do not require this step, because the conversion generates a digital value directly as the ADC reaches the approximation.

Analog Output

Analog output is the process of generating electrical signals from your computer. Analog output is generated by performing digital-to-analog (D/A) conversions. The available analog output types for a task are voltage and current. To perform a voltage or current task, a compatible device must be installed that can generate that form of signal.

Digital-to-analog conversion is the opposite of analog-to-digital conversion. In digital-to-analog conversion, the computer generates the data. The data might have been acquired earlier using analog input or may have been generated by software on the computer. A digital-to-analog converter (DAC) accepts this data and uses it to vary the voltage on an output pin over time. The DAC generates an analog signal that the DAC can send to other devices or circuits.

A DAC has an update clock that tells the DAC when to generate a new value. The function of the update clock is similar to the function of the sample clock for an ADC. At each cycle the clock, the DAC converts a digital value to an analog voltage and creates an output as a voltage on a pin. When used with a high speed clock, the DAC can create a signal that appears to vary constantly and smoothly.

Digital I/O

Digital signals are electrical signals that transfer digital data over a wire. These signals typically have only two states—on and off, also known as high and low, or 1 and 0. When sending a digital signal across a wire, the sender applies a voltage to the wire and the receiver uses the voltage level to determine the value being sent. The voltage ranges for each digital value depend on the voltage level standard being used. Digital signals have many uses; the simplest application of a digital signal is controlling or measuring digital or finite state devices such as switches and LEDs. Digital signals also can transfer data; you can use them to program devices or communicate between devices. In addition, you can use digital signals as clocks or triggers to control or synchronize other measurements.

You can use the digital lines in a DAQ device to acquire a digital value. This acquisition is based on software timing. On some devices, you can configure the lines individually to either measure or generate digital samples. Each line corresponds to a channel in the task.

You can use the digital port(s) in a DAQ device to acquire a digital value from a collection of digital lines. This acquisition is based on software timing. You can configure the ports individually to either measure or generate digital samples. Each port corresponds to a channel in the task.

Counters

A counter is a digital timing device. You typically use counters for event counting, frequency measurement, period measurement, position measurement, and pulse generation.

When you configure a counter for simple event counting, the counter increments when an active edge is received on the source. In order for the counter to increment on an active edge, the counter must be armed or started. A counter has a fixed number it can count to as determined by the resolution of the counter. For example, a 24-bit counter can count to:

$$2^{(\text{Counter Resolution})} - 1 = 2^{24} - 1 = 16,777,215$$

When a 24-bit counter reaches the value of 16,777,215, it has reached the terminal count. The next active edge forces the counter to roll over and start at 0.

Using DAQ Software

National Instruments data acquisition devices have a driver engine that communicates between the device and the application software. You can use LabVIEW to communicate with these driver engines. NI-DAQmx contains VIs, functions, and development tools for controlling measurement devices. Refer to *Programming with DAQ in LabVIEW* for more information about programming with DAQmx.

In addition, you can use MAX to configure your data acquisition devices. In this section, you learn about the driver engines and about using MAX to configure your data acquisition device.

DAQ Hardware Configuration

Before using a data acquisition device, you must confirm that the software can communicate with the device by configuring the devices. The devices are already configured for the computers in this class.

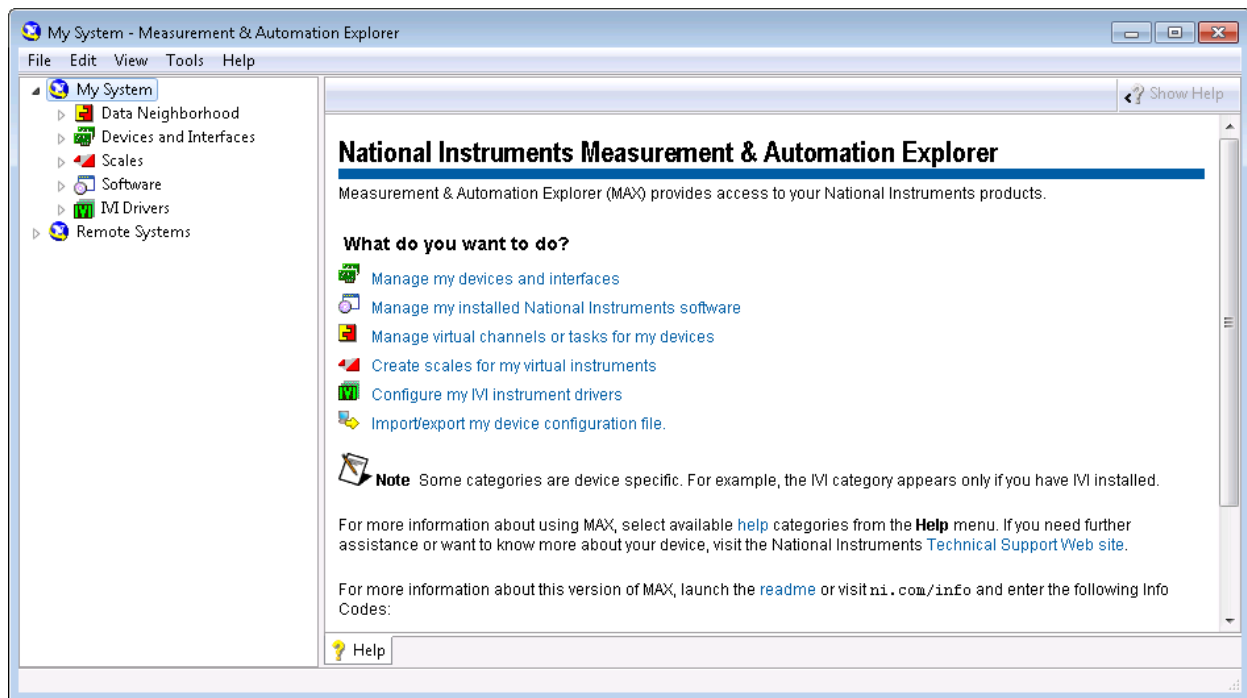
Windows

The Windows Configuration Manager keeps track of all the hardware installed in the computer, including National Instruments DAQ devices. If you have a Plug & Play (PnP) device, such as an E Series MIO device, the Windows Configuration Manager automatically detects and configures the device. If you have a non-PnP device, or legacy device, you must configure the device manually using the **Add New Hardware** option in the Windows Control Panel.

You can verify the Windows Configuration by accessing the Device Manager. You can see **Data Acquisition Devices**, which lists all DAQ devices installed in the computer. Double-click a DAQ device to display a dialog box with tabbed pages. The **General** tab displays overall information regarding the device. The **Driver** tab specifies the driver version and location for the DAQ device. The **Details** tab contains additional information about hardware configuration. The **Resources** tab specifies the system resources to the device such as interrupt levels, DMA, and base address for software-configurable devices.

Measurement & Automation Explorer

MAX establishes all device and channel configuration parameters. After installing a DAQ device in the computer, you must run this configuration utility. MAX reads the information the Device Manager records in the Windows Registry and assigns a logical device number to each DAQ device. Use the device number to refer to the device in LabVIEW. Access MAX by double-clicking the icon on the desktop or selecting **Tools»Measurement & Automation Explorer** in LabVIEW. The following window is the primary MAX window. MAX is also the means for SCXI and SCC configuration.

Figure 6-8. The Primary MAX Window

The device parameters that you can set using the configuration utility depend on the device. MAX saves the logical device number and the configuration parameters in the Windows Registry.

The plug and play capability of Windows automatically detects and configures switchless DAQ devices, such as the NI PCI-6024E, when you install a device in the computer.

Simulating a DAQ Device

You can create NI-DAQmx simulated devices in NI-DAQmx 7.4 or later. Using NI-DAQmx simulated devices, you can try NI products in your application without the hardware. When you later acquire the hardware, you can import the NI-DAQmx simulated device configuration to the physical device using the MAX Portable Configuration Wizard. With NI-DAQmx simulated devices, you also can export a physical device configuration onto a system that does not have the physical device installed. Then, using the NI-DAQmx simulated device, you can work on your applications on a portable system and upon returning to the original system, you can easily import your application work.

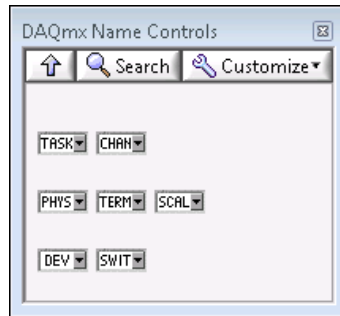
Programming with DAQ in LabVIEW

You already have an understanding of transducers, signals, DAQ device configuration, and MAX. Now you can begin learning to use LabVIEW to develop a data acquisition application. NI-DAQmx supports software packages other than LabVIEW, but this course describes only LabVIEW development of DAQ applications.

DAQmx Name Controls

The DAQmx Name Controls palette includes controls for the task name, channel name, physical channel, terminal, scale name, device name, and switch. You can also create these controls by right-clicking the corresponding input terminal on a DAQmx VI and selecting **Create»Control**. For more information on these controls, refer to the *NI-DAQmx Help*.

Figure 6-9. DAQmx Name Controls Palette



DAQmx - Data Acquisition VIs

Use the NI-DAQmx VIs with NI-DAQ hardware devices to develop instrumentation, acquisition, and control applications. Refer to the *DAQ Getting Started Guide* or the *NI-DAQ Readme* for a complete listing of devices that NI-DAQmx supports.

The DAQmx - Data Acquisition palette includes the following constants and VIs.

Constants

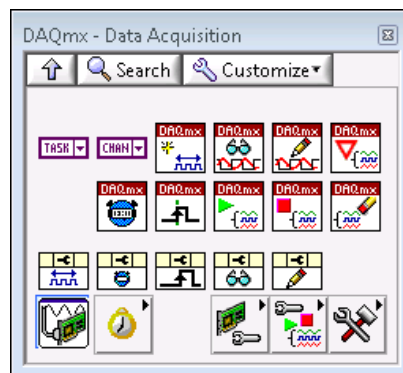
- **DAQmx Task Name Constant**—Lists all tasks you create and save using the DAQ Assistant. Right-click the constant and select I/O Name Filtering from the shortcut menu to limit the tasks the constant displays and to limit what you can enter in the constant.
- **DAQmx Global Channel Constant**—Lists all global channels you create and save by using the DAQ Assistant. Click and select Browse to select multiple channels. Right-click the constant and select I/O Name Filtering from the shortcut menu to limit the channels the constant displays and to limit what you can enter in the constant.

VIs

- **DAQmx Create Virtual Channel VI**—Creates a virtual channel or set of virtual channels and adds them to a task. The instances of this polymorphic VI correspond to the I/O type of the channel, such as analog input, digital output, or counter output; the measurement or generation to perform, such as temperature measurement, voltage generation, or event counting; and in some cases, the sensor to use, such as a thermocouple or RTD for temperature measurements. This function sets the same settings that you configure in MAX when creating a virtual channel. Use this function if the operator of your program might periodically change the physical channel connection, but not other important settings, such as terminal configuration mode or the custom scale applied. Use the physical channel pull-down menu to specify the device number of the DAQ device and the actual physical channel your signal is connected to.

- DAQmx Read VI—Reads samples from the task or channels you specify. The instances of this polymorphic VI specify what format of samples to return, whether to read a single sample or multiple samples at once, and whether to read from one or multiple channels. You can select an instance by right clicking on the DAQmx Read VI and choose Select Type to include additional configuration options for read operations.
- DAQmx Write VI—Writes samples to task or channels you specify. The instances of this polymorphic VI specify the format of the samples to write, whether to write one or multiple samples, and whether to write to one or multiple channels. You can select an instance by right clicking on the DAQmx Write VI and choose Select Type to include additional configuration options for write operations
- DAQmx Wait Until Done VI—Waits for the measurement or generation to complete. Use this VI to ensure that the specified operation is complete before you stop the task.
- DAQmx Timing VI—Configures number of samples to acquire or generate and creates a buffer when needed. The instances of this polymorphic VI correspond to the type of timing to use on the task.
- DAQmx Trigger VI—Configures triggering for the task. The instances of this polymorphic VI correspond to the trigger and trigger type to configure.
- DAQmx Start Task VI—Transitions the task to the running state to begin the measurement or generation. Using this VI is required for some applications and is optional for others.
- DAQmx Stop Task VI—Stops the task and returns it to the state the task was in before you used the DAQmx Start Task VI or used the DAQmx Write VI with the autostart input set to TRUE.
- DAQmx Clear Task VI—Clears the task. Before clearing, this VI stops the task, if necessary, and releases any resources reserved by the task. You cannot use a task after you clear it unless you recreate the task.

Figure 6-10. DAQmx Data Acquisition Palette



D. Controlling Instruments

When you use a PC to automate a test system, you are not limited to the type of instrument you can control. LabVIEW allows you to control virtually any type of instrument including amplifiers, analyzers, calibrators, oscilloscopes, power supplies, switches, signal generators, sensors, and more.

You can mix and match instruments from various categories. The most common categories of instrument interfaces are GPIB, USB, Ethernet, serial, and modular instruments. Additional types of instruments include image acquisition, motion control, parallel port, NI-CAN, and other devices. This course focuses on GPIB.

When you use PCs to control instruments, you need to understand properties of the instrument, such as the communication protocols to use and the command set that the device expects. Refer to the instrument programmers manual or user manual for information about the properties of an instrument.

Benefits of Instrument Control

LabVIEW makes it easy to control and acquire data from any instrument over any bus. You can automate measurements from several devices, analyze data as you acquire it, and then log the results to disk. This automation can save enormous amounts of time and money. Also, using a single application to control multiple tasks and instruments allows you to consolidate and synchronize those tasks.

LabVIEW has many tools that help simplify programming for instrument control, including instrument drivers, examples, wizards, and Express VIs. These tools can help you do more with your time by reducing repetitive time-consuming tasks, leaving you to focus on developing the unique parts of your application. You can download instrument drivers for almost any instrument free of charge.

GPIB

The ANSI/IEEE Standard 488.1-1987, also known as General Purpose Interface Bus (GPIB), describes a standard interface for communication between instruments and controllers from various vendors. GPIB instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications. GPIB instruments are often used as stand-alone bench-top instruments where measurements are taken by hand. You can automate these measurements by using a PC to control the GPIB instruments.

IEEE 488.1 contains information about electrical, mechanical, and functional specifications. The ANSI/IEEE Standard 488.2-1992 extends IEEE 488.1 by defining a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands.

GPIB is a digital, 8-bit parallel communication interface with data transfer rates of 1 Mbyte/s and higher, using a three-wire handshake. The bus supports one system controller, usually a computer, and up to 14 additional instruments.

The GPIB protocol categorizes devices as controllers, talkers, or listeners to determine which device has active control of the bus. Each device has a unique GPIB primary address between 0 and 30. The Controller defines the communication links, responds to devices that request service, sends GPIB commands, and passes/receives control of the bus. Controllers instruct Talkers to talk and to place data on the GPIB. You can address only one device at a time to talk. The Controller addresses the Listener to listen and to read data from the GPIB. You can address several devices to listen.

Data Transfer Termination

Termination informs listeners that all data has been transferred. You can terminate a GPIB data transfer in the following three ways:

- The GPIB includes an End Or Identify (EOI) hardware line that can be asserted with the last data byte. This is the preferred method.
- Place a specific end-of-string (EOS) character at the end of the data string itself. Some instruments use this method instead of or in addition to the EOI line assertion.
- The listener counts the bytes transferred by handshaking and stops reading when the listener reaches a byte count limit. This method is often a default termination method because the transfer stops on the logical OR of EOI, EOS (if used) in conjunction with the byte count. As a precaution, the byte count on the listener is often set higher than the expected byte count so as not to miss any samples.

Data Transfer Rate

To achieve the high data transfer rate that the GPIB was designed for, you must limit the number of devices on the bus and the physical distance between devices.

You can obtain faster data rates with HS488 devices and controllers. HS488 is an extension to GPIB that most NI controllers support.



Note Refer to the National Instruments GPIB support Web site at ni.com/support/gpibsupp.htm for more information about GPIB.

Configuring a GPIB Instrument

The software architecture for instrument control using LabVIEW is similar to the architecture for DAQ. Instrument interfaces such as GPIB include a set of drivers. Use MAX to configure the interface.



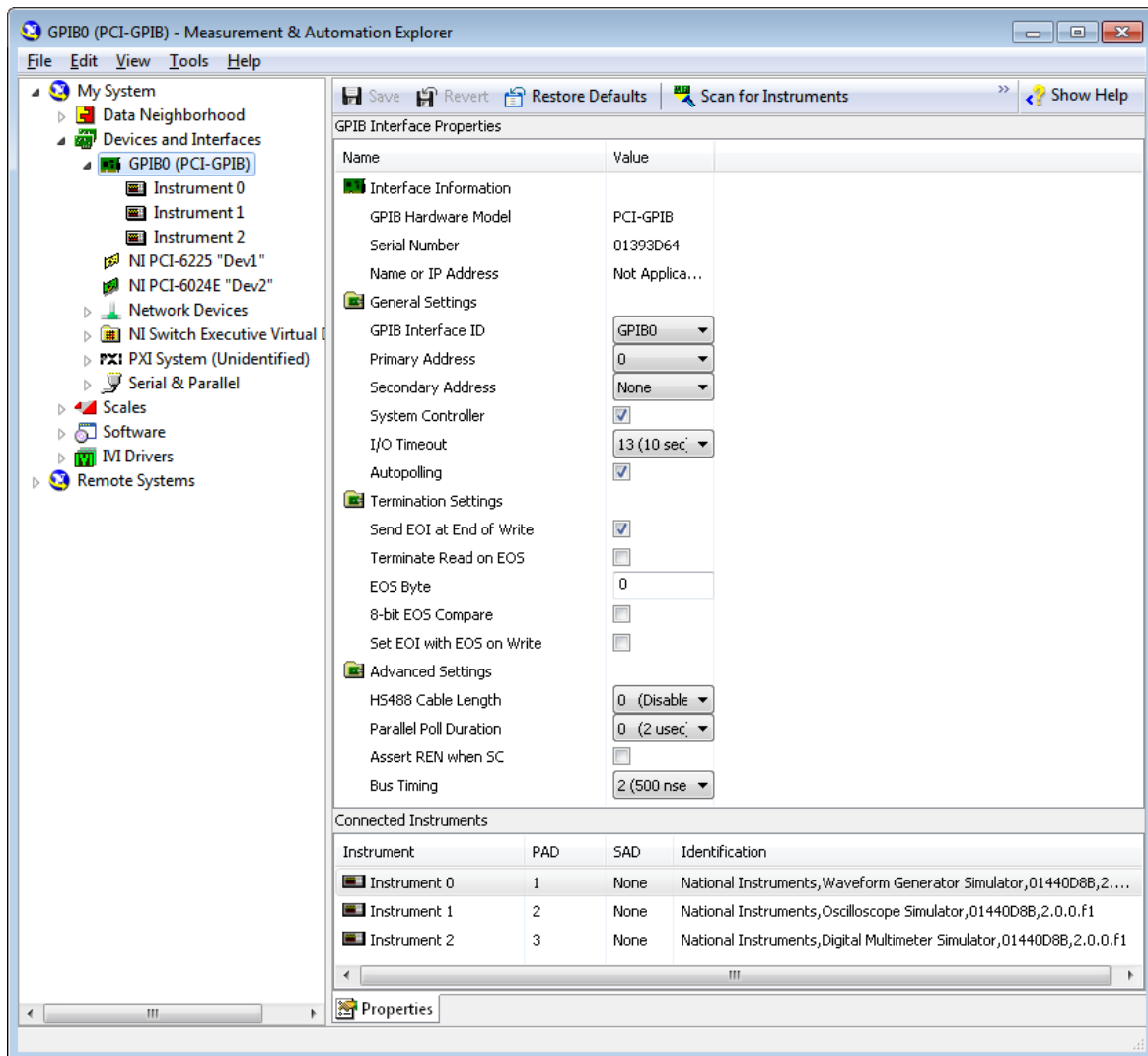
Note Most GPIB drivers are available for download at ni.com/support/gpib/versions.htm. Always install the newest version of these drivers unless otherwise instructed in the release notes.

MAX (Windows: GPIB)

Use MAX to configure and test the GPIB interface. MAX interacts with the various diagnostic and configuration tools installed with the driver and also with the Windows Registry and Device Manager. The driver-level software is in the form of a DLL and contains all the functions that directly communicate with the GPIB interface. The Instrument I/O VIs and functions directly call the driver software.

Open MAX by double-clicking the icon on the desktop or by selecting **Tools»Measurement & Automation Explorer** in LabVIEW. The following example shows a GPIB interface in MAX after clicking the **Scan For Instruments** button on the toolbar.

Figure 6-11. GPIB Interface in Measurement & Automation Explorer

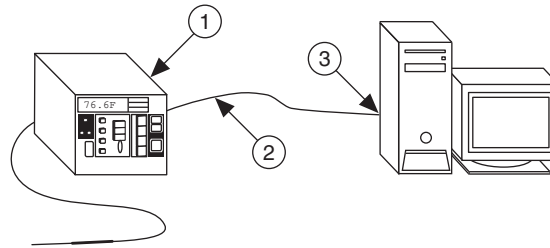


Configure the objects listed in MAX by right-clicking each item and selecting an option from the shortcut menu.

Serial

Serial communication transmits data between a computer and a peripheral device, such as a programmable instrument or another computer. Serial communication uses a transmitter to send data one bit at a time over a single communication line to a receiver. Use this method when data transfer rates are low or you must transfer data over long distances. Most computers have one or more serial ports, so you do not need any extra hardware other than a cable to connect the instrument to the computer or to connect two computers to each other.

Figure 6-12. Serial Instrument Example



1 RS-232 Instrument	2 RS-232 Cable	3 Serial Port
---------------------	----------------	---------------

You must specify four parameters for serial communication: the baud rate of the transmission, the number of data bits that encode a character, the sense of the optional parity bit, and the number of stop bits. A character frame packages each transmitted character as a single start bit followed by the data bits.

Baud rate is a measure of how fast data moves between instruments that use serial communication.

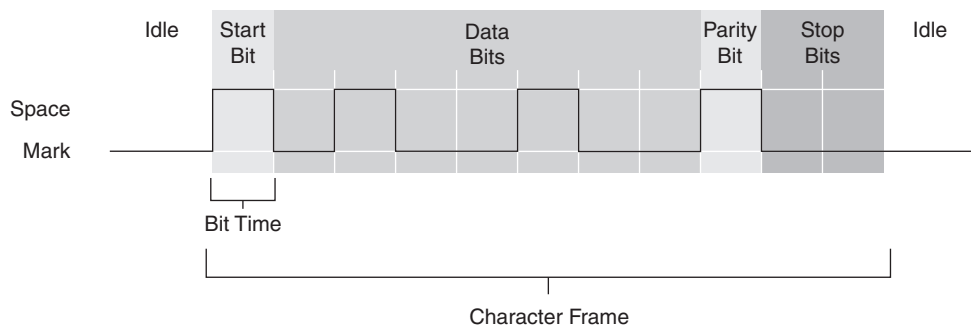
Data bits are transmitted upside down and backwards, which means that inverted logic is used and the order of transmission is from least significant bit (LSB) to most significant bit (MSB). To interpret the data bits in a character frame, you must read from right to left and read 1 for negative voltage and 0 for positive voltage.

An optional parity bit follows the data bits in the character frame. The parity bit, if present, also follows inverted logic. This bit is included as a means of error checking. You specify ahead of time for the parity of the transmission to be even or odd. If you choose for the parity to be odd, the parity bit is set in such a way so the number of 1s add up to make an odd number among the data bits and the parity bit.

The last part of a character frame consists of 1, 1.5, or 2 stop bits that are always represented by a negative voltage. If no further characters are transmitted, the line stays in the negative (MARK) condition. The transmission of the next character frame, if any, begins with a start bit of positive (SPACE) voltage.

The following figure shows a typical character frame encoding the letter m.

Figure 6-13. Character Frame for the Letter m



RS-232 uses only two voltage states, called MARK and SPACE. In such a two-state coding scheme, the baud rate is identical to the maximum number of bits of information, including control bits, that are transmitted per second.

MARK is a negative voltage, and SPACE is positive. The previous illustration shows how the idealized signal looks on an oscilloscope. The following is the truth table for RS-232:

Signal > +3 V = 0

Signal < -3 V = 1

The output signal level usually swings between +12 V and -12 V. The dead area between +3 V and -3 V is designed to absorb line noise.

A start bit signals the beginning of each character frame. It is a transition from negative (MARK) to positive (SPACE) voltage. Its duration in seconds is the reciprocal of the baud rate. If the instrument is transmitting at 9,600 baud, the duration of the start bit and each subsequent bit is about 0.104 ms. The entire character frame of eleven bits would be transmitted in about 1.146 ms.

Interpreting the data bits for the transmission yields 1101101 (binary) or 6D (hex). An ASCII conversion table shows that this is the letter m.

This transmission uses odd parity. There are five ones among the data bits, already an odd number, so the parity bit is set to 0.

Data Transfer Rate

You can calculate the maximum transmission rate in characters per second for a given communication setting by dividing the baud rate by the bits per character frame.

In the previous example, there are a total of eleven bits per character frame. If the transmission rate is set at 9,600 baud, you get $9,600/11 = 872$ characters per second. Notice that this is the maximum character transmission rate. The hardware on one end or the other of the serial link might not be able to reach these rates, for various reasons.

Serial Port Standards

The following examples are the most common recommended standards of serial port communication:

- **RS232 (ANSI/EIA-232 Standard)** is used for many purposes, such as connecting a mouse, printer, or modem. It also is used with industrial instrumentation. Because of improvements in line drivers and cables, applications often increase the performance of RS232 beyond the distance and speed in the standards list. RS232 is limited to point-to-point connections between PC serial ports and devices.
- **RS422 (AIA RS422A Standard)** uses a differential electrical signal as opposed to the unbalanced (single-ended) signals referenced to ground with RS232. Differential transmission, which uses two lines each to transmit and receive signals, results in greater noise immunity and longer transmission distances as compared to RS232.
- **RS485 (EIA-485 Standard)** is a variation of RS422 that allows you to connect up to 32 devices to a single port and define the necessary electrical characteristics to ensure adequate signal voltages under maximum load. With this enhanced multidrop capability, you can create networks of devices connected to a single RS485 serial port. The noise immunity and multidrop capability make RS485 an attractive choice in industrial applications that require many distributed devices networked to a PC or other controller for data collection and other operations.

Instrument Control Programming

VISA is a high-level API that calls low-level drivers. VISA can control instruments over GPIB, USB, Ethernet, serial, and other interfaces as it makes the appropriate driver calls depending on the type of instrument used. When debugging VISA problems, remember that an apparent VISA problem could be an installation problem with one of the drivers that VISA calls.

In LabVIEW, VISA is a single library of functions you use to communicate with a variety of different I/O interfaces that any instrument may provide. Because of this, you do not need to use separate I/O palettes to program an instrument. For example, some devices offer multiple interface types such as Ethernet, USB, and GPIB on the same instrument. If the LabVIEW instrument driver were written with functions on the **Instrument I/O»GPIB** palette, those instrument driver VIs would not work for the instrument with the USB interface. VISA solves this problem by providing a single set of functions that work for any type of interface. Therefore, many LabVIEW instrument drivers use VISA as the I/O language.

VISA Programming Terminology

The following terminology is similar to that used for instrument driver VIs:

- **Resource**—Any instrument in the system, including serial and parallel ports.
- **Session**—You must open a VISA session to a resource to communicate with it, similar to a communication channel. When you open a session to a resource, LabVIEW returns a VISA session number, which is a unique refnum to that instrument. You must use the session number in all subsequent VISA functions.

- **Instrument Descriptor**—Exact name of a resource. The descriptor specifies the interface type (GPIB, USB, TCP/IP, ASRL), the address of the device (logical address or primary address), and the VISA session type (INSTR or Event).

The instrument descriptor is similar to a telephone number, the resource is similar to the person with whom you want to speak, and the session is similar to the telephone line. Each call uses its own line, and crossing these lines results in an error. Table 6-1 shows the proper syntax for the instrument descriptor for a few different interfaces.

Table 6-1. Syntax for Various Instrument Interfaces

Interface	Syntax
Asynchronous serial	ASRL [device] [: : INSTR]
GPIB	GPIB [device] : : primary address [: : secondary address] [: : INSTR]
VXI instrument through embedded or MXIbus controller	VXI [device] : : VXI logical address [: : INSTR]
GPIB-VXI controller	GPIB-VXI [device] [: : GPIB-VXI primary address] : : VXI logical address [: : INSTR]

You can use an alias you assign in MAX instead of the instrument descriptor.

The most commonly used VISA communication functions are the VISA Write and VISA Read functions. Most instruments require you to send information in the form of a command or query before you can read information back from the instrument. Therefore, the VISA Write function is usually followed by a VISA Read function. The VISA Write and VISA Read functions work with any type of instrument communication and are the same whether you are doing GPIB or serial communication. However, because serial communication requires you to configure extra parameters, you must start the serial port communication with the VISA Configure Serial Port VI.

Using Instrument Drivers

Imagine the following scenario. You wrote a LabVIEW VI that communicates with a specific oscilloscope in your lab. Unfortunately, the oscilloscope no longer works, and you must replace it. However, this particular oscilloscope is no longer made. You found a different brand of oscilloscope that you want to purchase, but your VI no longer works with the new oscilloscope. You must rewrite your VI.

When you use an instrument driver, the driver contains the code specific to the instrument. Therefore, if you change instruments, you must replace only the instrument driver VIs with the instrument driver VIs for the new instrument, which greatly reduces your redevelopment time. Instrument drivers help make test applications easier to maintain because the drivers contain all the I/O for an instrument in one library, separate from other code. When you upgrade hardware, upgrading the application is easier because the instrument driver contains all the code specific to that instrument.

Understanding Instrument Drivers

A LabVIEW Plug and Play instrument driver is a set of VIs that control a programmable instrument. Each VI corresponds to an instrument operation, such as configuring, triggering, and reading measurements from the instrument. Instrument drivers help users get started using instruments from a PC and saves them development time and cost because users do not need to learn the programming protocol for each instrument. With open-source, well-documented instrument drivers, you can customize their operation for better performance. A modular design makes the driver easier to customize.

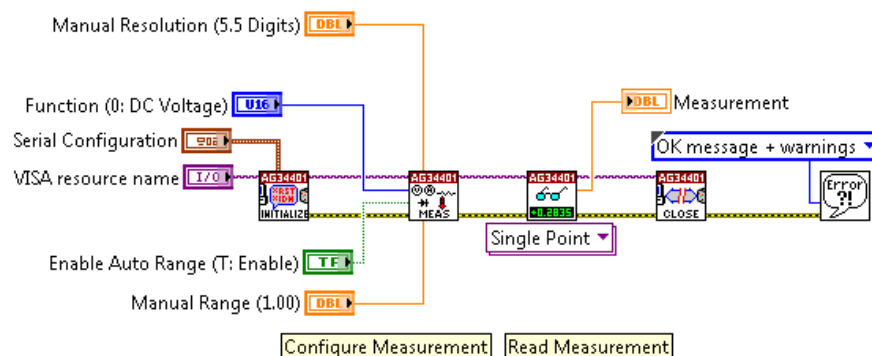
Locating Instrument Drivers

You can locate most LabVIEW Plug and Play instrument drivers in the Instrument Driver Finder. You can access the Instrument Driver Finder within LabVIEW by selecting **Tools» Instrumentation»Find Instrument Drivers** or **Help»Find Instrument Drivers**. The Instrument Driver Finder pulls drivers from NI's extensive Instrument Driver Network that can be found at ni.com/idnet. When you install an instrument driver, an example program using the driver is added to the NI Example Finder.

Example Instrument Driver VI

The block diagram in Figure 6-14 initializes the Agilent 34401 digital multimeter (DMM), uses a configuration VI to choose the resolution and range, select the function, and enable or disable auto range, uses a data VI to read a single measurement, closes the instrument, and checks the error status. Every application that uses an instrument driver has a similar sequence of events: Initialize, Configure, Read Data, and Close.

Figure 6-14. Agilent 34401 DMM Instrument Driver Example



This is an example program that is available in the NI Example Finder when you install the LabVIEW Plug and Play instrument driver for the Agilent 34401 DMM.

Many programmable instruments have a large number of functions and modes. With this complexity, it is necessary to provide a consistent design model that aids both instrument driver developers as well as end users who develop instrument control applications.

The VIs in an instrument driver are organized into six categories. These categories are summarized in the following table.

Category	Description
Initialize	The Initialize VI establishes communication with the instrument and is the first instrument driver VI called.
Configure	Configure VIs are software routines that configure the instrument to perform specific operations. After calling these VIs, the instrument is ready to take measurements or stimulate a system.
Action/Status	Action/Status VIs command the instrument to carry out an action (for example, arming a trigger) or obtain the current status of the instrument or pending operations.
Data	The data VIs transfer data to or from the instrument.
Utility	Utility VIs perform a variety of auxiliary operations, such as reset and self-test.
Close	The close VI terminates the software connection to the instrument. This is the last instrument driver VI called.

Self-Review: Quiz

1. After opening a file, which output does the Open/Create/Replace File function return?
 - a. File path
 - b. File name
 - c. Refnum out
 - d. Task out

2. What is Measurement & Automation Explorer (MAX)?
 - a. A tool to configure and test DAQ devices
 - b. A tool to test instrument communication
 - c. A configurable Express VI
 - d. A window to view LabVIEW project files

3. Which of the following are benefits of instrument control?
 - a. Automate processes
 - b. Save time
 - c. One platform for multiple tasks
 - d. Limited to only one type of instrument

4. VISA is a high-level API that calls low-level drivers.
 - a. True
 - b. False

Self-Review: Quiz Answers

1. After opening a file, which output does the Open/Create/Replace File function return?
 - a. File path
 - b. File name
 - c. Refnum out**
 - d. Task out

2. What is Measurement & Automation Explorer (MAX)?
 - a. A tool to configure and test DAQ devices**
 - b. A tool to test instrument communication**
 - c. A configurable Express VI
 - d. A window to view LabVIEW project files

3. Which of the following are benefits of instrument control?
 - a. Automate processes**
 - b. Save time**
 - c. One platform for multiple tasks**
 - d. Limited to only one type of instrument

4. VISA is a high-level API that calls low-level drivers.
 - a. True**
 - b. False

Notes

Using Sequential and State Machine Algorithms

The first step in developing a LabVIEW application is to explore the architectures that exist within LabVIEW. Architectures are essential for creating a successful software design. The most common architectures are usually grouped into design patterns.

As a design pattern gains acceptance, it becomes easier to recognize when a design pattern has been used. This recognition helps you and other developers read and modify VIs that are based on design patterns.

There are many design patterns for LabVIEW VIs. Most applications use at least one design pattern. In this course, you explore the State Machine design pattern. Learn more about design patterns in the *LabVIEW Core 2* course.

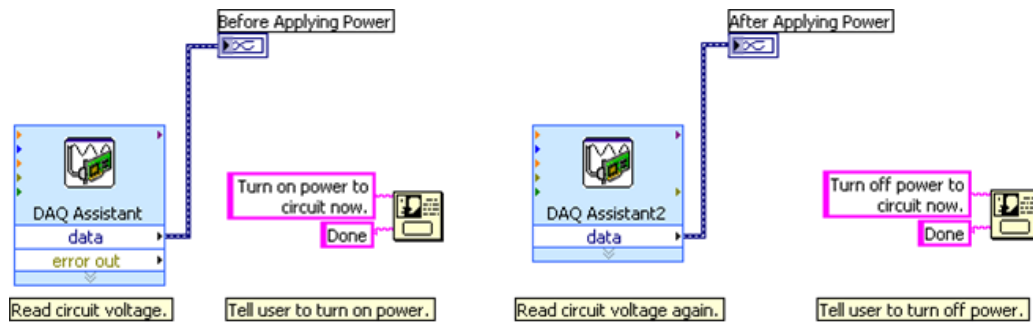
Topics

- A. Using Sequential Programming
- B. Using State Programming
- C. State Machines

A. Using Sequential Programming

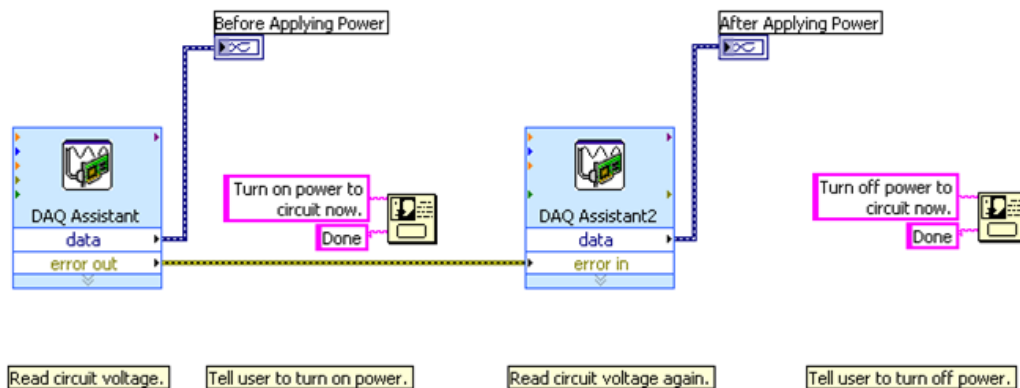
Many of the VIs you write in LabVIEW accomplish sequential tasks. How you program these sequential tasks can be very different. Consider the block diagram in Figure 7-1. In this block diagram, a voltage signal is acquired, a dialog box prompts the user to turn on the power, the voltage signal is acquired again, and the user is prompted to turn off the power. However, in this example, there is nothing in the block diagram to force the execution order of these events. Any one of these events could happen first.

Figure 7-1. Unsequenced Tasks

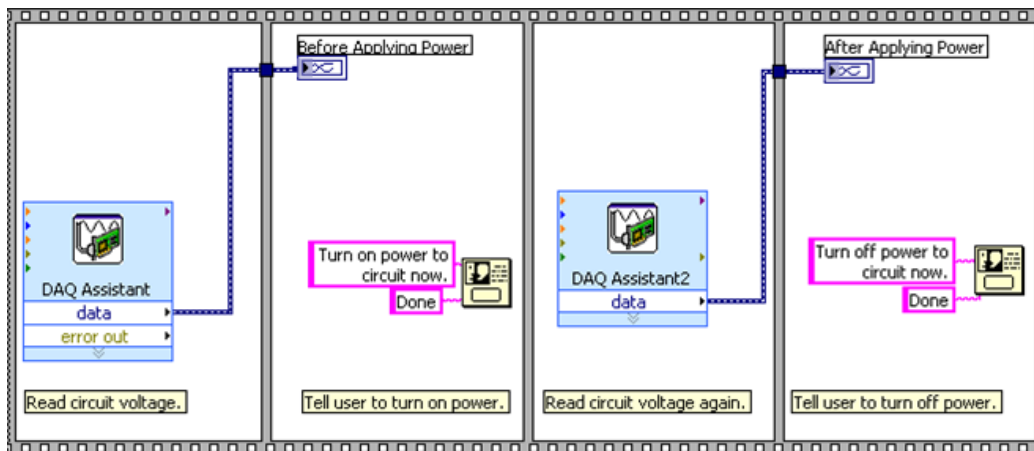


In LabVIEW, you can complete sequential tasks by placing each task in a separate subVI, and wiring the subVIs in the order you want them to execute using the error cluster wires. However, in this example, only two of the tasks have an error cluster. Using the error clusters, you can force the execution order of the two DAQ Assistants, but not the One Button Dialog functions, as shown in Figure 7-2.

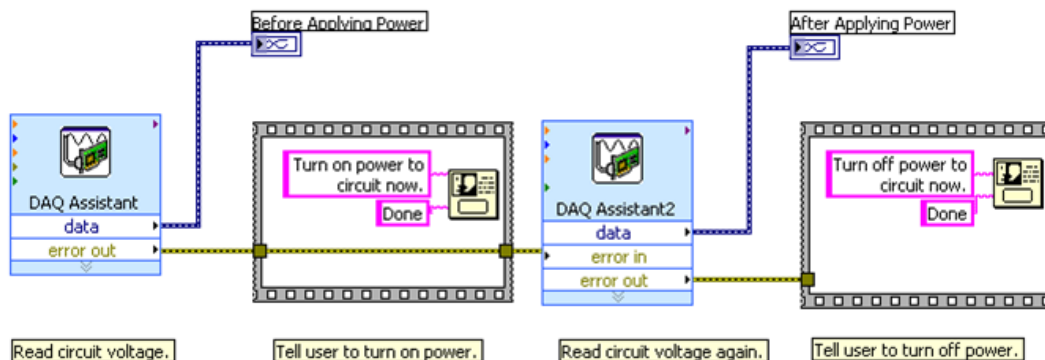
Figure 7-2. Partially Sequenced Tasks



You can use a Sequence structure to force the order of operations of block diagram objects. A Sequence structure contains one or more subdiagrams, or frames, that execute in sequential order; a frame cannot begin execution until everything in the previous frame has completed execution. Figure 7-3 shows an example of this VI using a Sequence structure to force execution order.

Figure 7-3. Tasks Sequenced with a Sequence Structure

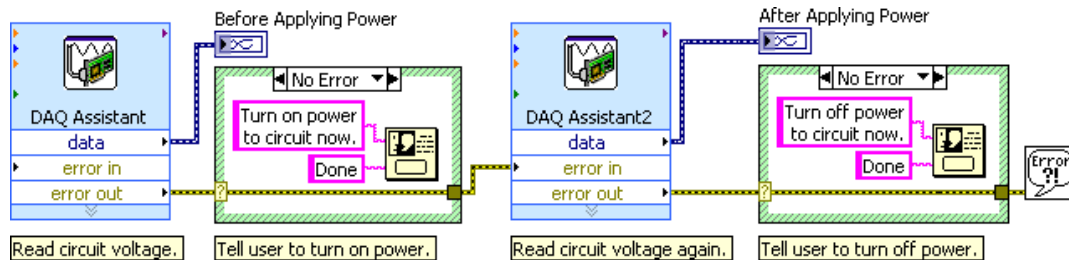
To take advantage of the inherent parallelism in LabVIEW, avoid overusing Sequence structures. Sequence structures guarantee the order of execution, but prohibit parallel operations. Another negative to using Sequence structures is that you cannot stop the execution part way through the sequence. A good way to use Sequence structures for this example is shown in Figure 7-4.

Figure 7-4. Tasks Sequenced with Sequence Structures and an Error Cluster

The best way to write this VI, however, is to enclose the One Button Dialog functions in a Case structure, and wire the error cluster to the case selector.

Use Sequence structures sparingly because they do not enforce error checking and will continue to go through a sequence even after errors are detected. Rely on data flow rather than sequence structures to control the order of execution.

Figure 7-5. Tasks Sequenced with an Error Cluster and Case Structures



B. Using State Programming

Although a Sequence structure and sequentially wired subVIs both accomplish the task, often your VIs require more complex programming:

- What if you must change the order of the sequence?
- What if you must repeat an item in the sequence more often than the other items?
- What if some items in the sequence execute only when certain conditions are met?
- What if you must stop the program immediately, rather than waiting until the end of the sequence?

Although your program may not have any of the above requirements, there is always the possibility that the program must be modified in the future. For this reason, a state programming architecture is a good choice, even if a sequential programming structure would be sufficient.

C. State Machines

The state machine design pattern is a common and very useful design pattern for LabVIEW. You can use the state machine design pattern to implement any algorithm that can be explicitly described by a state diagram or flowchart. A state machine usually implements a moderately complex decision-making algorithm, such as a diagnostic routine or a process monitor.

A *state machine*, which is more precisely defined as a finite state machine, consists of a set of states and a transition function that maps to the next state. Finite state machines have many variations. The two most common finite state machines are the Mealy machine and the Moore machine. A Mealy machine performs an action for each transition. A Moore machine performs a specific action for each state in the state transition diagram. The state machine design pattern template in LabVIEW implements any algorithm described by a Moore machine.

Applying State Machines

Use state machines in applications where distinguishable states exist. Each state can lead to one or multiple states or end the process flow. A state machine relies on user input or in-state calculation to determine which state to go to next. Many applications require an initialization state, followed by a default state, where many different actions can be performed. The actions performed can depend on previous and current inputs and states. A shutdown state commonly performs clean up actions.

State machines are commonly used to create user interfaces. In a user interface, different user actions send the user interface into different processing segments. Each processing segment acts as a state in the state machine. Each segment can lead to another segment for further processing or wait for another user action. In a user interface application, the state machine constantly monitors the user for the next action to take.

Process testing is another common application of the state machine design pattern. For a process test, a state represents each segment of the process. Depending on the result of each state's test, a different state might be called. This can happen continually, resulting in an in-depth analysis of the process you are testing.

The advantage of using a state machine is that after you have created a state transition diagram, you can create LabVIEW VIs easily.

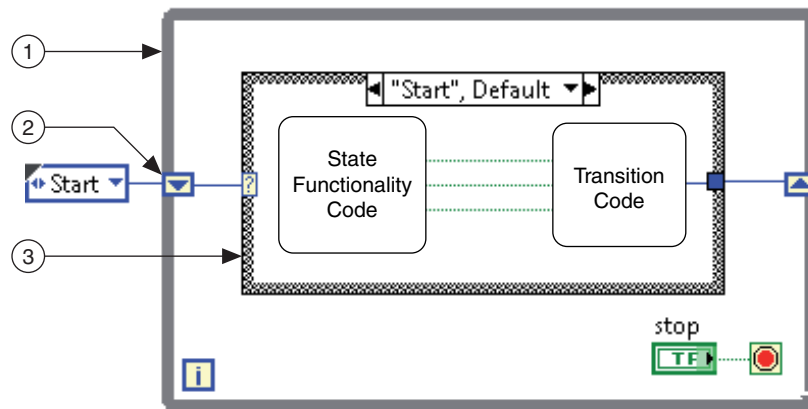
State Machine Infrastructure

Translating the state transition diagram into a LabVIEW block diagram requires the following infrastructure components:

- **While Loop**—Continually executes the various states
- **Case Structure**—Contains a case for each state and the code to execute for each state
- **Shift Register**—Contains state transition information
- **State Functionality Code**—Implements the function of the state
- **Transition Code**—Determines the next state in the sequence

Figure 7-6 shows the basic structure of a state machine implemented in LabVIEW for a temperature data acquisition system.

Figure 7-6. Basic Infrastructure of a LabVIEW State Machine



1 While Loop

2 Shift Register

3 Case Structure

The flow of the state transition diagram is implemented by the While Loop. The individual states are represented by cases in the Case structure. A shift register on the While Loop keeps track of the current state and communicates the current state to the Case structure input.

Controlling State Machines

The best method for controlling the initialization and transition of state machines is the enumerated type control. Enums are widely used as case selectors in state machines. However, if the user attempts to add or delete a state from the enumerated type control, the remaining wires that are connected to the copies of this enumerated type control break. This is one of the most common obstacles when implementing state machines with enumerated type controls. One solution to this problem is to create a type defined enumerated type control. This causes all the enumerated type control copies to automatically update if you add or remove a state.

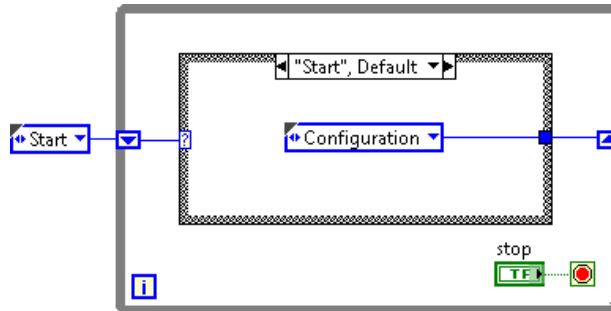
Transitioning State Machines

There are many ways to control what case a Case structure executes in a state machine. Choose the method that best suits the function and complexity of your state machine. Of the methods to implement transitions in state machines, the most common and easy to use is the single Case structure transition code, which can be used to transition between any number of states. This method provides for the most scalable, readable, and maintainable state machine architecture. The other methods can be useful in specific situations, and it is important for you to be familiar with them.

Default Transition

For the default transition, no code is needed to determine the next state, because there is only one possible state that occurs next. Figure 7-7 shows a design pattern that uses a default transition implemented for a temperature data acquisition system.

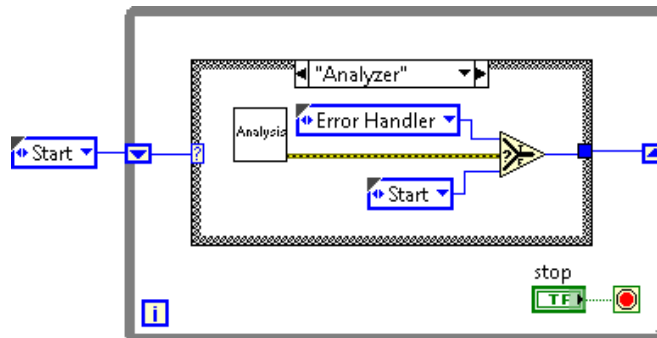
Figure 7-7. Single Default Transition



Transition Between Two States

The following method involves making a decision on a transition between two states. There are several patterns commonly used to accomplish this. Figure 7-8 shows the Select function used to transition between two states.

Figure 7-8. Select Function Transition Code



This method works well if you know that the individual state always transitions between two states. However, this method limits the scalability of the application. If you need to modify the state to transition among more than two states, this solution would not work and would require a major modification of the transition code.

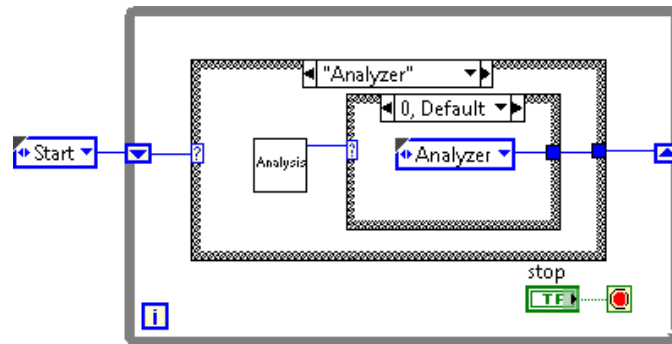
Transition Among Two or More States

Create a more scalable architecture by using one of the following methods to transition among states.

- **Case Structure**—Use a Case structure instead of the Select function for the transition code.

Figure 7-9 shows the Case structure transition implemented for a temperature data acquisition system.

Figure 7-9. Case Structure Transition Code

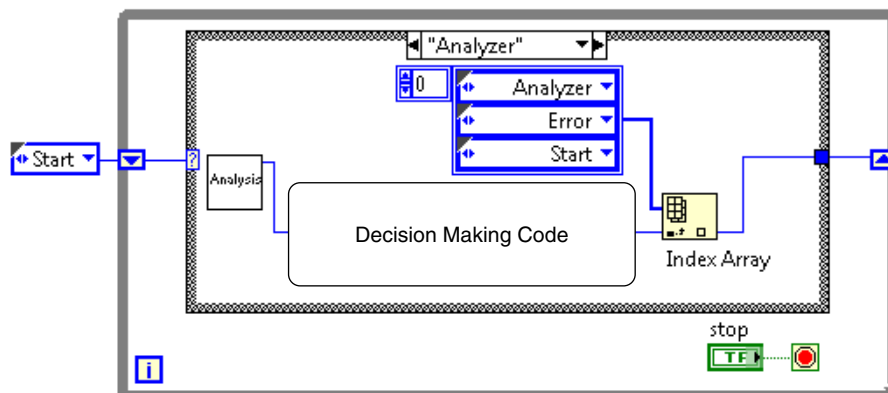


One advantage to using a Case structure is that the code is self-documenting. Because each case in the Case structure corresponds to an item in the enum, it is easy to read and understand the code. A Case structure also is scalable. As the application grows, you can add more transitions to a particular state by adding more cases to the Case structure for that state. A disadvantage to using a Case structure is that not all the code is visible at once. Because of the nature of the Case structure, it is not possible to see at a glance the complete functionality of the transition code.

- **Transition Array**—If you need more of the code to be visible than a Case structure allows, you can create a transition array for all the transitions that can take place.

Figure 7-10 shows the transition array implemented for a temperature data acquisition system.

Figure 7-10. Transition Array Transition Code

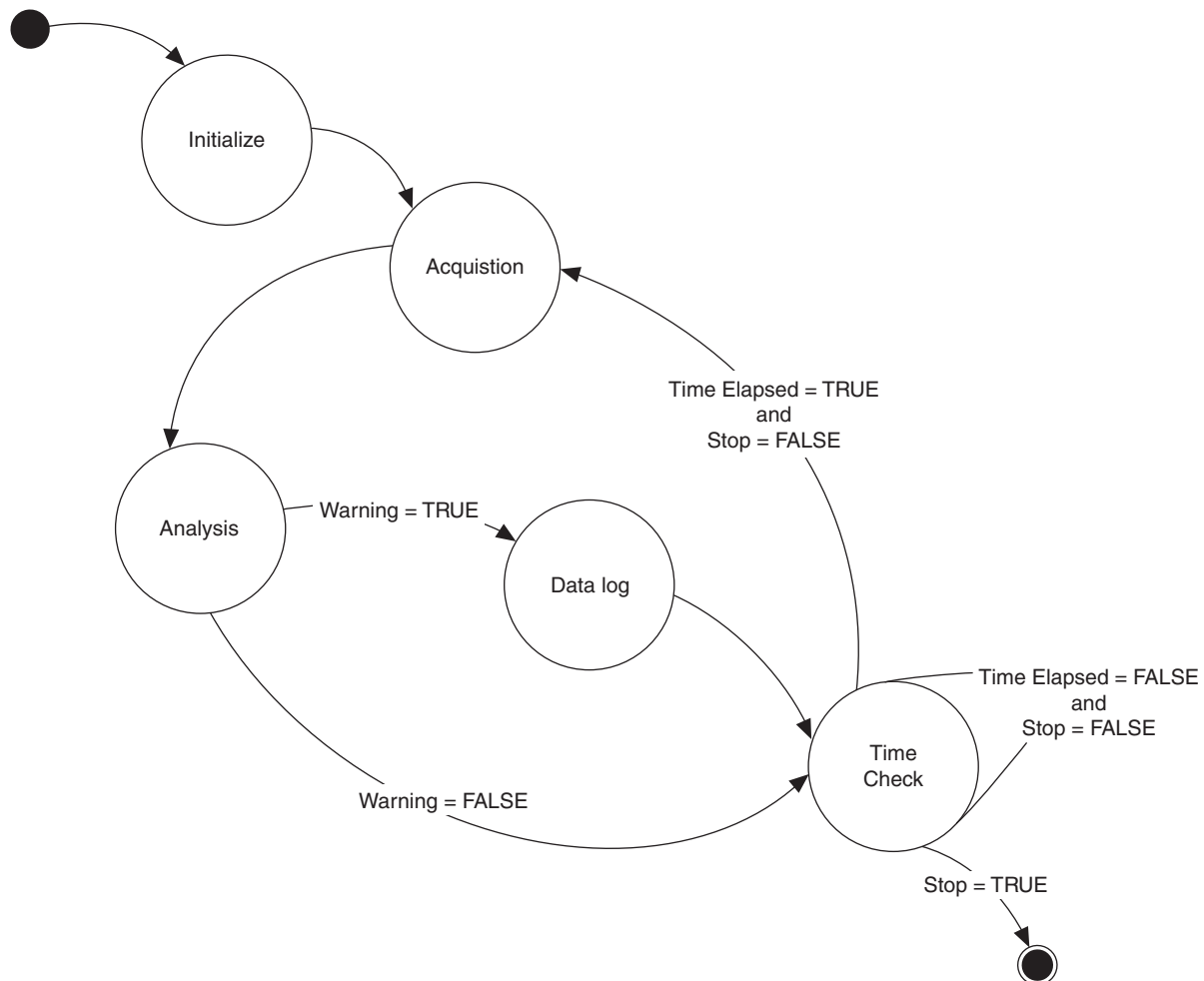


In this example, the decision making code provides the index that describes the next state. For example, if the code should progress to the Error Handler state next, the decision making code outputs the number 1 to the index input of the Index Array function. This design pattern makes the transition code scalable and easy to read. One disadvantage of this pattern is that you must use caution when developing the transition code because the array is zero-indexed.

Case Study: Course Project

The course project acquires a temperature every half second, analyzes each temperature to determine if the temperature is too high or too low, and alerts the user if there is a danger of heatstroke or freeze. The program logs the data if a warning occurs. If the user has not clicked the stop button, the entire process repeats. Figure 7-11 shows the state transition diagram for the course project.

Figure 7-11. State Transition Diagram for the Course Project



Figures 7-12 through 7-15 illustrate the states of the state machine that implements the state transition diagram detailed in Figure 7-11.

Figure 7-12. Acquisition State

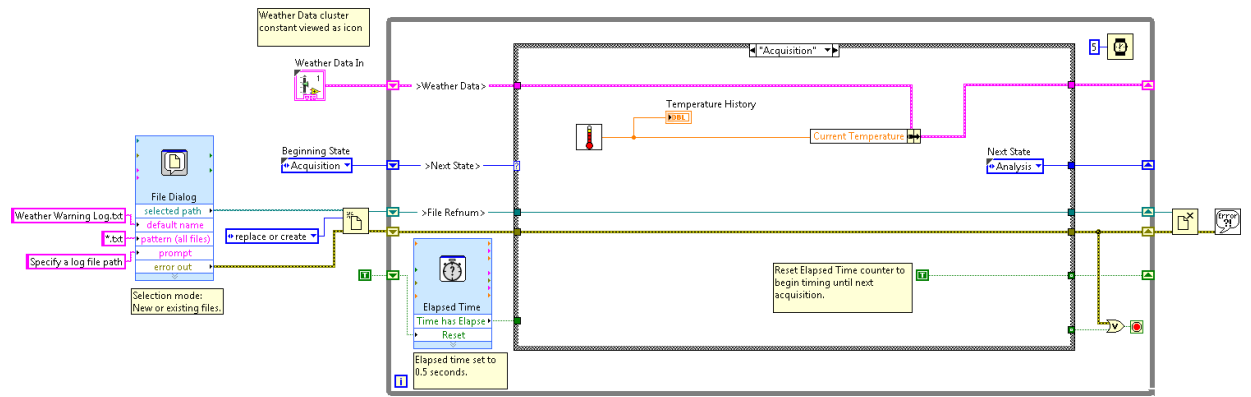


Figure 7-13. Analysis State

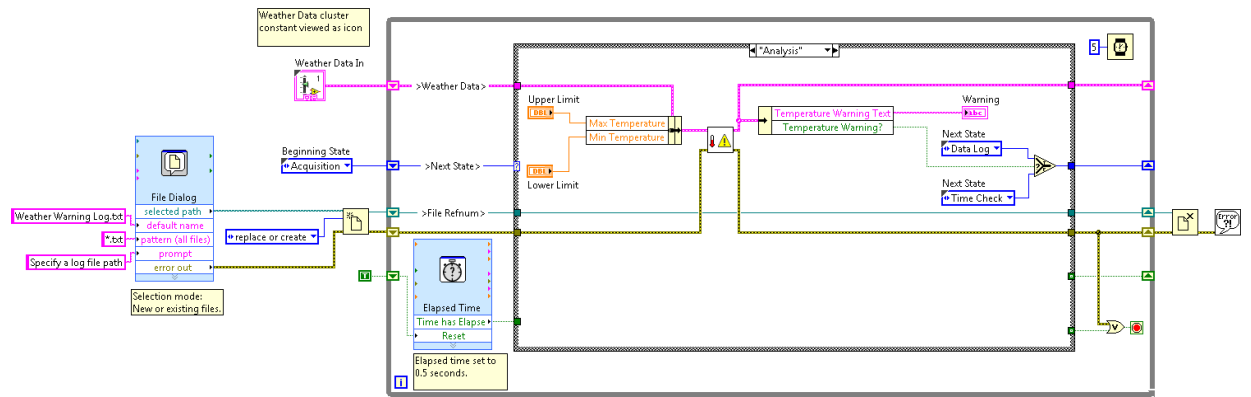


Figure 7-14. Data Log State

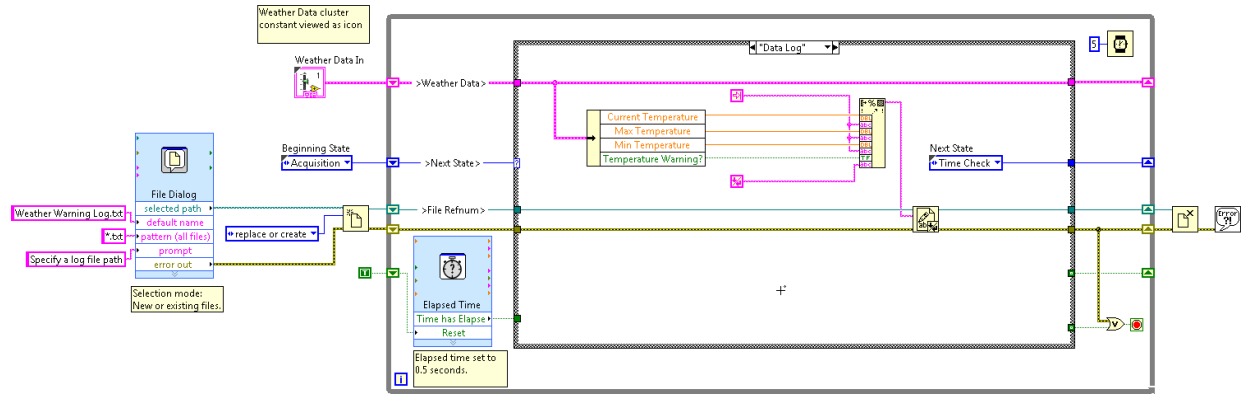
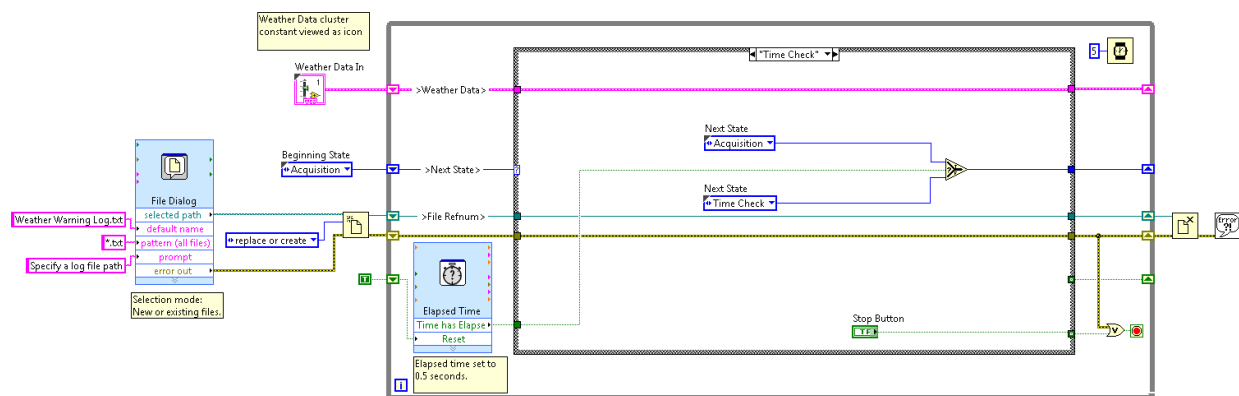


Figure 7-15. Time Check State



Self-Review: Quiz

1. When using a Sequence structure, you can stop the execution in the middle of a sequence.
 - a. True
 - b. False

2. Which of the following are benefits of using a state machine instead of a sequential structure?
 - a. You can change the order of the sequence.
 - b. You can repeat individual items in the sequence.
 - c. You can set conditions to determine when an item in the sequence should execute.
 - d. You can stop the program at any point in the sequence.

Self-Review: Quiz Answers

1. When using a Sequence structure, you can stop the execution in the middle of a sequence.
 - a. True
 - b. False**

2. Which of the following are benefits of using a state machine instead of a sequential structure?
 - a. You can change the order of the sequence.**
 - b. You can repeat individual items in the sequence.**
 - c. You can set conditions to determine when an item in the sequence should execute.**
 - d. You can stop the program at any point in the sequence.**

Notes

Solving Dataflow Challenges with Variables

LabVIEW is a dataflow language. As such, you generally use dataflow design principles in developing your code. However, in a few situations, such as communicating between two parallel loops, you must “break” the dataflow execution model by passing information between locations in the code that you cannot connect with a wire.

In this lesson you learn about variables, in general, and local variables, specifically, to handle special use cases where the dataflow execution model does not work. You also learn about the programming issues involved when using variables and how to overcome these challenges.

Because variables are not inherently part of the LabVIEW dataflow execution model, use variables carefully. Misusing variables can make the code hard to read, lead to unexpected behavior in VIs, and slow performance.

Topics

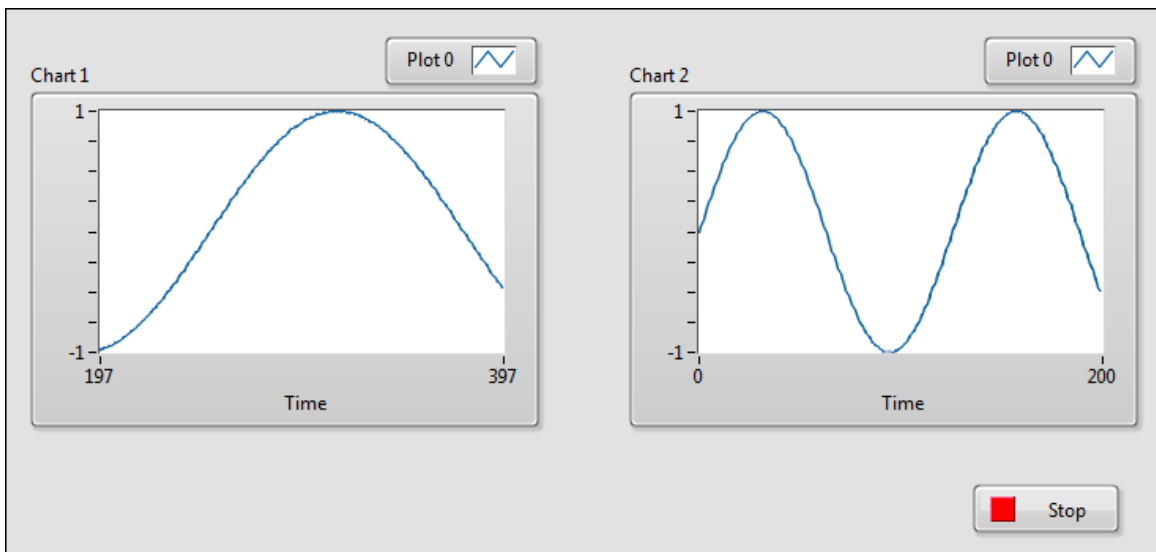
- A. Communicating Between Parallel Loops
- B. Writing to Controls and Reading from Indicators
- C. Variables
- D. Race Conditions

A. Communicating Between Parallel Loops

In this course, parallelism refers to executing multiple tasks at the same time. Consider the example of creating and displaying two sine waves at different frequencies. Using parallelism, you place one sine wave in a loop, and the second sine wave in a different loop.

A challenge in programming parallel tasks is passing data among multiple loops without creating a data dependency. For example, if you pass the data using a wire, the loops are no longer parallel. In the multiple sine wave example, you may want to share a single stop mechanism between the loops, as shown in Figure 8-1.

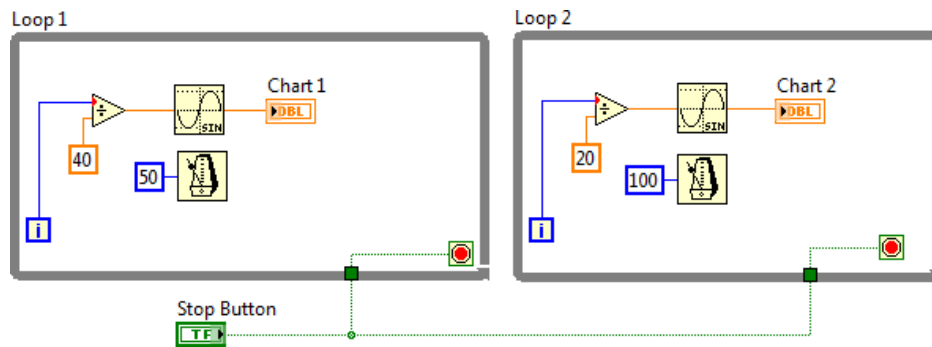
Figure 8-1. Parallel Loops Front Panel



Examine what happens when you try to share data among parallel loops with a wire using those different methods.

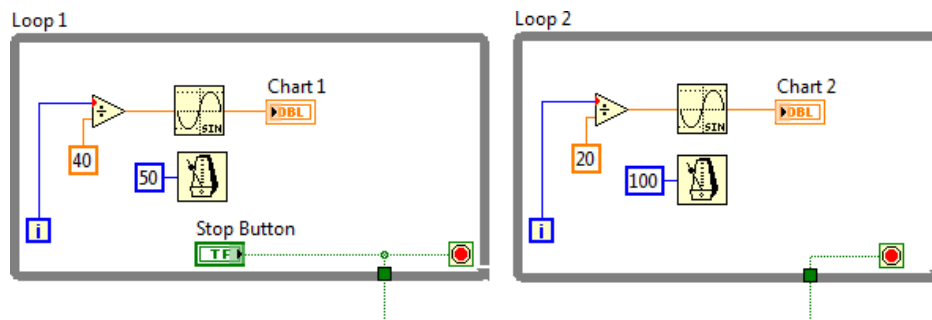
Method 1 (Incorrect)

Place the **Stop Button** terminal outside of both loops and wire it to each conditional terminal, as shown in Figure 8-2. The Loop control is a data input to both loops, therefore the **Stop Button** terminal is read only once, before either While Loop begins executing. If False is passed to the loops, the While Loops run indefinitely. Clicking the **Stop** button does not stop the VI because the button click is not read during the iteration of either loop.

Figure 8-2. Parallel Loops Method 1 Example

Method 2 (Incorrect)

Move the **Stop Button** terminal inside Loop 1 so that it is read in each iteration of Loop 1, as shown in the following block diagram. Although Loop 1 terminates properly, Loop 2 does not execute until it receives all its data inputs. Loop 1 does not pass data out of the loop until the loop stops, so Loop 2 must wait for the final value of the **Stop Button**, 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 True value from the **Stop Button** switch in Loop 1.

Figure 8-3. Parallel Loops Method 2 Example

Method 3 (Solution)

If you could read the value of the loop control from a file, you would no longer have a dataflow dependency between the loops, as each loop can independently access the file. In addition, reading and writing to files can be time consuming, at least in processor time.

Another way to accomplish the task of communicating between parallel loops is to find the location where the loop control data is stored in memory and read that memory location directly. A local variable allows you to do this by reading from the same front panel control in more than one location on the diagram. Later in this lesson you learn how to create and use a local variable that communicates between two parallel loops.

B. Writing to Controls and Reading from Indicators

In addition to communicating between two simultaneously running loops, there are other dataflow challenges that can be overcome with local variables.

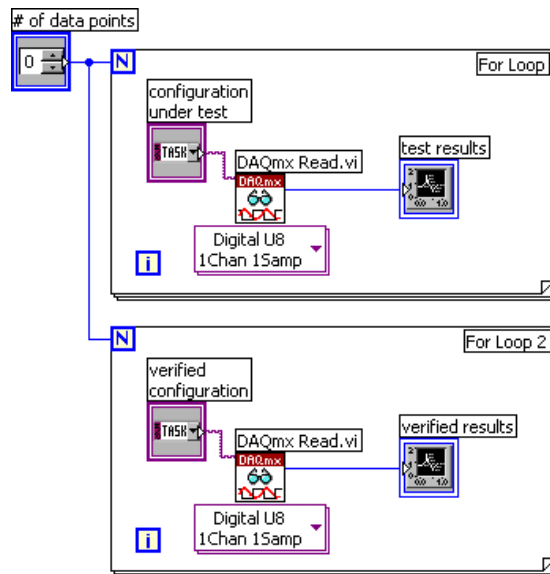
You can use a local variable to update a single front panel indicator from more than one location on the block diagram. For example, if you have a VI with two While Loops, you can update a front panel indicator to display which loop is currently executing.

With a local variable, you can write to or read from a control or indicator on the front panel. In effect, with a local variable you can access a front panel object as both an input and an output. For example, if the user interface requires users to log in, you can clear the **Login** and **Password** prompts each time a new user logs in. Use a local variable to read from the **Login** and **Password** string controls when a user logs in and to write empty strings to these controls when the user logs out.

C. Variables

In LabVIEW, the flow of data rather than the sequential order of commands determines the execution order of block diagram elements. Therefore, you can create block diagrams that have simultaneous operations. For example, you can run two For Loops simultaneously and display the results on the front panel, as shown in Figure 8-4.

Figure 8-4. Displaying the Results of Two For Loops on the Front Panel

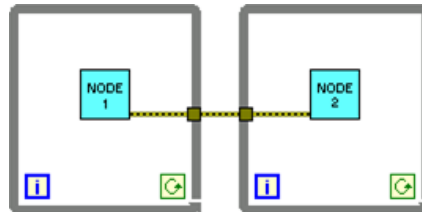


In LabVIEW, the flow of data rather than the sequential order of commands determines the execution order of block diagram elements. Therefore, you can create block diagrams that have simultaneous operations. For example, you can run two For Loops simultaneously and display the results on the front panel, as shown in the following block diagram. However, if you use wires to pass data between parallel loops, they no longer operate in parallel. Parallel loop operations can be

two parallel loops on the same block diagram without any data flow dependency, or two loops in separate VIs that are called at the same time.

The block diagram in Figure 8-5 does not run the two loops in parallel because of the wire between the two subVIs.

Figure 8-5. Data Dependency Imposed by Wire



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

In LabVIEW, *variables* are block diagram elements that allow you to access or store data in another location. The actual location of the data varies depending on the type of the variable. Local variables store data in front panel controls and indicators. Global variables and single-process shared variables store data in special repositories that you can access from multiple VIs. Functional global variables store data in While Loop shift registers. Regardless of where the variable stores data, all variables allow you to circumvent normal data flow by passing data from one place to another without connecting the two places with a wire. For this reason, variables are useful in parallel architectures, but also have certain drawbacks, such as race conditions.

Local Variables—Using Variables in a Single VI

Local variables transfer data within a single VI.

In LabVIEW, you read data from or write data to a front panel object using its block diagram terminal. However, a front panel object has only one block diagram terminal, and your application might need to access the data in that terminal from more than one location.

Local and global variables pass information between locations in the application that you cannot connect with a wire. Use local variables to access front panel objects from more than one location in a single VI. Use global variables to access and pass data among several VIs.

Use a Feedback Node to store data from a previous VI or loop execution.

Creating Local Variables

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

You also can select a local variable from the **Functions** palette and place it on the block diagram. The local variable node is not yet associated with a control or indicator.



To associate a local variable with a control or indicator, right-click the local variable node and select **Select Item** from the shortcut menu. The expanded shortcut menu lists all the front panel objects that have owned labels.

LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels.

Reading and Writing to Variables

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

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

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

On the block diagram, you can distinguish read variables from write variables the same way you distinguish controls from indicators. A read variable has a thick border similar to a control. A write variable has a thin border similar to an indicator.

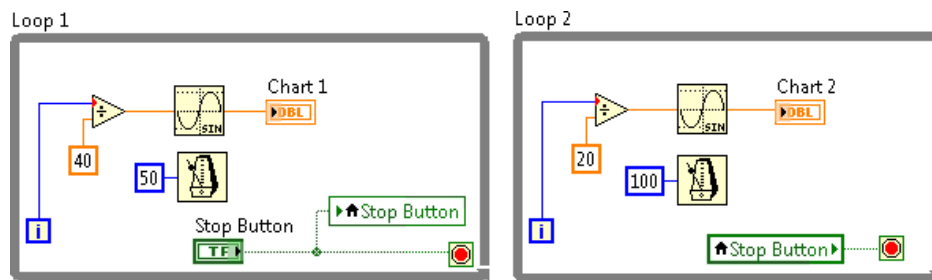
Local Variable Example

In the *Communicating Between Parallel Loops* section of this lesson, you saw an example of a VI that used parallel loops. The front panel contained a single switch that stopped the data generation displayed on two graphs. On the block diagram, the data for each chart is generated within an individual While Loop to allow for separate timing of each loop. The Loop Control terminal stopped both While Loops. In this example, the two loops must share the switch to stop both loops at the same time.

For both charts to update as expected, the While Loops must operate in parallel. Connecting a wire between While Loops to pass the **Stop Button** data makes the While Loops execute serially, rather

than in parallel. Figure 8-6 shows a block diagram of this VI using a local variable to pass the **Stop Button** data.

Figure 8-6. Local Variable Used to Stop Parallel Loops



Loop 2 reads a local variable associated with the **Stop Button**. When you set the button to False on the front panel, the terminal in Loop 1 writes a False value to the **Stop Button** local variable and to the conditional terminal in Loop 1. Loop 2 reads the **Stop Button** local variable and writes a False value to the Loop 2 conditional terminal. Thus, the loops run in parallel and terminate simultaneously when you turn off a single front panel switch.

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

For example, if the user interface requires users to log in, you can clear the **Login** and **Password** prompts each time a new user logs in. Use a local variable to read from the **Login** and **Password** string controls when a user logs in and to write empty strings to these controls when the user logs out.

Global and Shared Variables—Using Variables Among VIs

You also can use variables to access and pass data among several VIs that run simultaneously. A local variable shares data within a VI. A global variable also shares data, but it shares data among multiple VIs. For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You can use a global variable to terminate both loops with a single Boolean control. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

You also can use a single-process shared variable in the same way you use a global variable. A shared variable is similar to a local variable or a global variable, but allows you to share data across a network. A shared variable can be single-process or network-published. Although network-published shared variables are beyond the scope of this course, by using the single-process shared variable, you can later change to a network-published shared variable.

Use a global variable to share data among VIs on the same computer, especially if you do not use a project file. Use a single-process shared variable if you may need to share the variable information among VIs on multiple computers in the future.

Creating Global Variables

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

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

Select a global variable from the **Functions** palette and place it on the block diagram.



Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel.

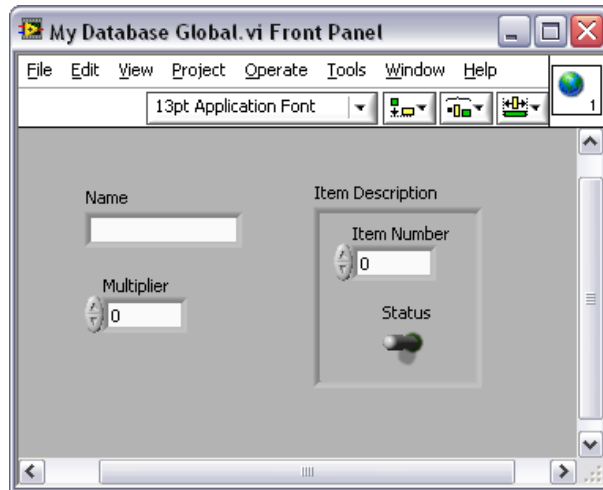
LabVIEW uses owned labels to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

You can create several single variable global VIs, each with one front panel object, or if you want to group similar variables together, you can create one global variable VI with multiple front panel objects.

A global VI with multiple objects is more efficient because you can group related variables together. The block diagram of a VI can include several global variable nodes that are associated with controls and indicators on the front panel of a global VI. You place global VIs on other VIs the same way you place subVIs on other VIs. Each time you place a new global variable node on a block diagram, LabVIEW creates a new global VI associated only with that global variable node and copies of it.

Figure 8-7 shows a global VI front panel window with a numeric, a string, and a cluster containing a numeric and a Boolean control. The toolbar does not show the **Run**, **Stop**, or related buttons as a normal front panel window.

Figure 8-7. Global VI Front Panel Window



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

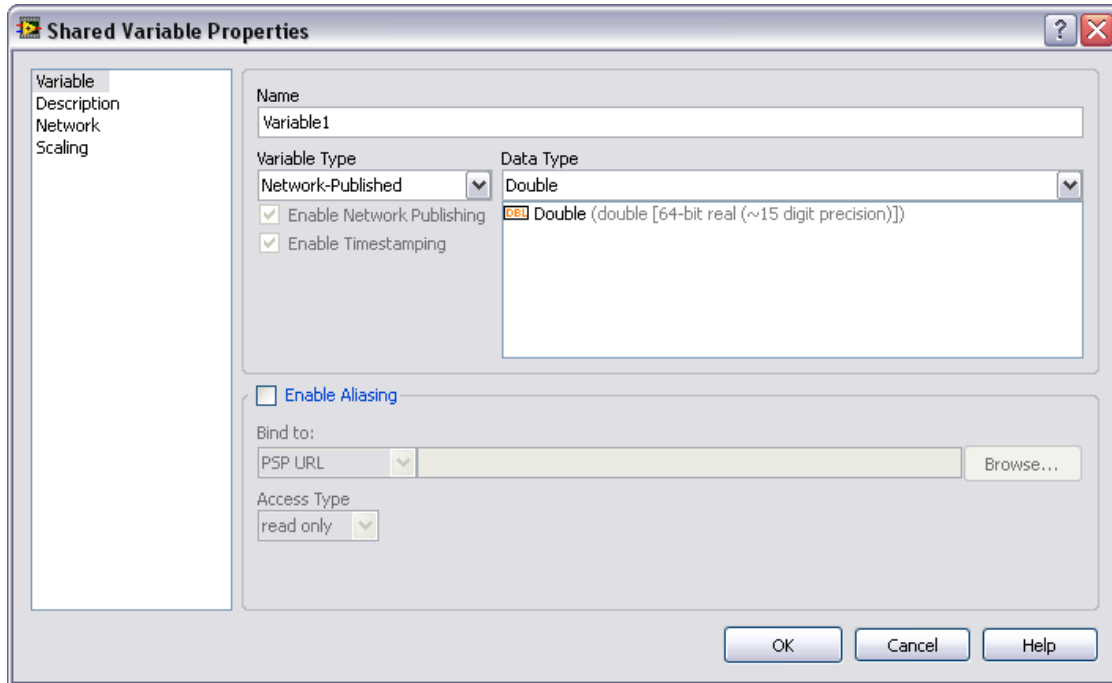
You also can use the Operating tool or Labeling tool to click the global variable node and select the front panel object from the shortcut menu.

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

Creating Single-Process Shared Variables

You must use a project file to use a shared variable. To create a single-process shared variable, right-click **My Computer** in the **Project Explorer** window and select **New»Variable**. The **Shared Variable Properties** dialog box appears, as shown in Figure 8-8.

Figure 8-8. Shared Variable Properties Dialog Box



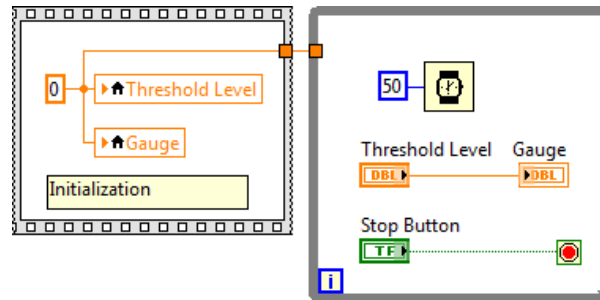
Under **Variable Type**, select **Single Process**. Give the variable a name and a data type. After you create the shared variable, it automatically appears in a new library in your project file. Save the library. You can add additional shared variables to this library as needed. You can drag and drop the variable from the listing in the **Project Explorer** window directly to the block diagram. Use the short-cut menu to switch between writing or reading. Use the error clusters on the variable to impose data flow.

Initializing Variables

To initialize a local or global variable, verify that the variable contains known data values before the VI runs. Otherwise, the variables might contain data that causes the VI to behave incorrectly. If the variable relies on a computation result for the initial value, make sure LabVIEW writes the value to the variable before it attempts to access the variable for any other action. Wiring the write action in parallel with the rest of the VI can cause a race condition.

To make sure it executes first, you can isolate the code that writes the initial value for the variable to the first frame of a sequence structure or to a subVI and wire the subVI to execute first in the data flow of the block diagram, as shown in Figure 8-9.

Figure 8-9. Initializing Variables



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

Avoiding Programming Issues Inherent with Variables

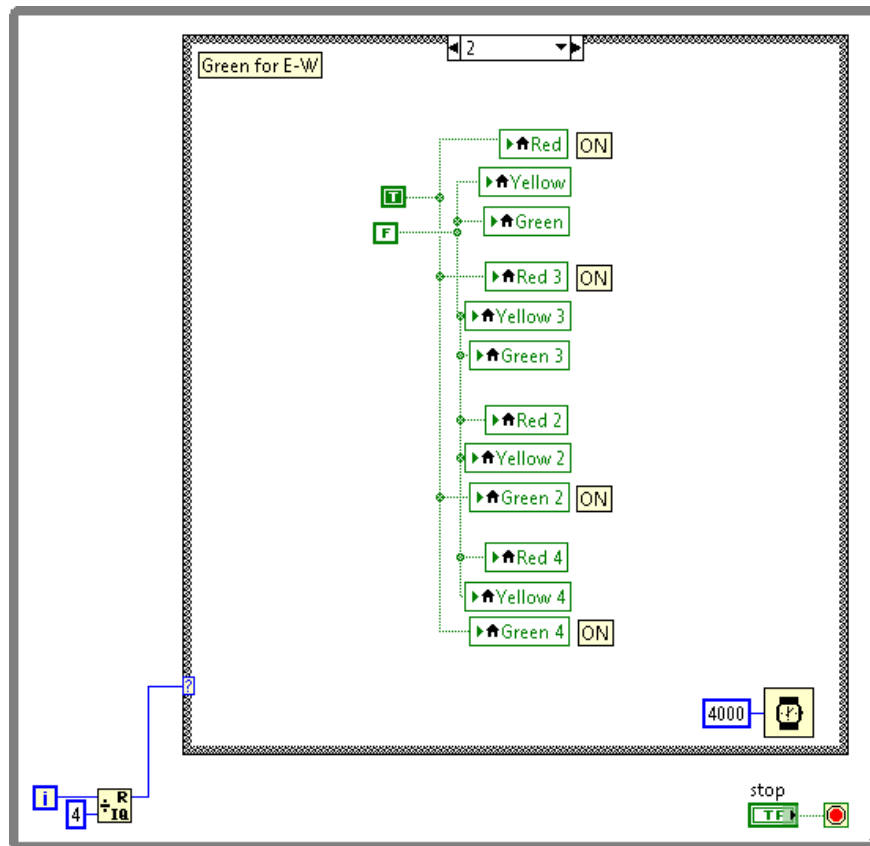
Local and global variables are advanced LabVIEW concepts. They are inherently not part of the LabVIEW dataflow execution model. Some of the programming issues with variables include the following:

- Make the block diagram code hard to read—Because variables break the dataflow model, you cannot use wires to follow the flow of data.
- Lead to unexpected behavior in VIs—Using variables instead of a connector pane or using variables to access values in each frame of a sequence structure is a bad practice and can cause unexpected behavior in VIs.
- Slow performance—Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of data flow, slows performance.

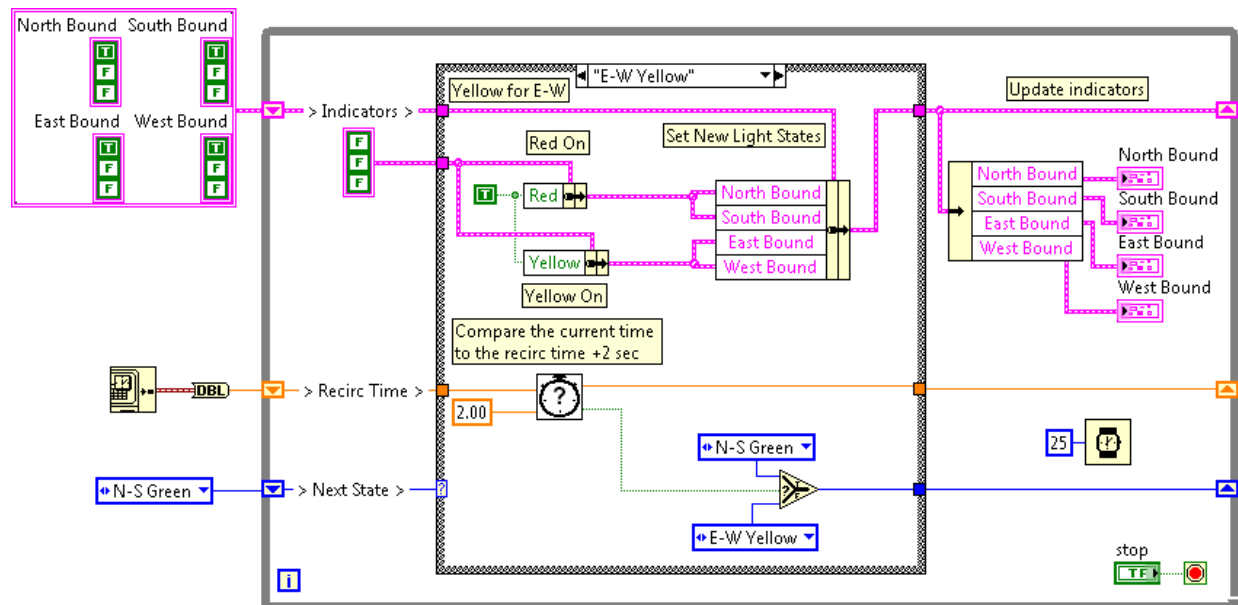
Figure 8-10 and Figure 8-11 illustrate two ways to program the same example. Figure 8-10 illustrates a poor use of variables while Figure 8-11 illustrates a more optimal design. The programming example is a traffic light application implemented as a state machine. Each state updates the lights for the next stage of the light sequence.

In the state shown in Figure 8-10, the east and west traffic has a green light, while the north and south traffic has a red light. This stage waits for 4 seconds, as shown by the Wait (ms) function.

Figure 8-10. Too Many Variables Used



The example shown in Figure 8-11 accomplishes the same task but more efficiently and using a better design. Notice that this example is much easier to read and understand than the previous example, mostly by reducing variable use. By placing the indicators in the While Loop outside the Case structure, the indicators can update after every state without using a variable. This example is less difficult to modify for further functionality, such as adding left turn signals, than the previous example.

Figure 8-11. Reduced Variables

D. Race Conditions

A race condition occurs when the timing of events or the scheduling of tasks unintentionally affects an output or data value. Race conditions are a common problem for programs that execute multiple tasks in parallel and share data between them using variables.

Race conditions are difficult to identify and debug, because the outcome depends upon the order in which the operating system executes scheduled tasks and the timing of external events. Often, code with a race condition can return the same result thousands of times in testing, but still be capable of returning different result at any time.

The best way to avoid race conditions is by using the following techniques:

- Controlling and limiting shared resources.
- Properly sequencing instructions by specifying an execution order.
- Reducing the use of variables.

Controlling and Limiting Shared Resources

Race conditions are most common when two tasks have both read and write access to a resource. A resource is any entity that is shared between the processes. When dealing with race conditions, the most common shared resources are data storage, such as variables. Other examples of resources include files and references to hardware resources.

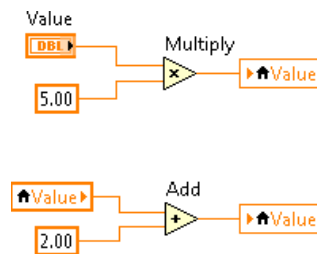
When you alter a resource from multiple locations, you introduce the possibility of a race condition. Therefore, an ideal way to avoid race conditions is to minimize shared resources and the number of writers to the remaining shared resources. In general, it is not harmful to have multiple

readers or monitors for a shared resource. However, try to use only one writer or controller for a shared resource to avoid race conditions.

Properly Sequencing Instructions

Code in which data flow is not properly used to control the execution order can cause some race conditions. When a data dependency is not established, LabVIEW can schedule tasks in any order, which creates the possibility for race conditions if the tasks depend upon each other. Consider the example in Figure 8-12.

Figure 8-12. Simple Race Condition



The code in this example has four possible outcomes, depending on the order in which the operations execute.

Outcome 1: $\text{Value} = (\text{Value} \times 5) + 2$

1. Terminal reads Value.
2. $\text{Value} \times 5$ is stored in Value.
3. Local variable reads $\text{Value} \times 5$.
4. $(\text{Value} \times 5) + 2$ is stored in Value.

Outcome 2: $\text{Value} = (\text{Value} + 2) \times 5$

1. Local variable reads Value.
2. $\text{Value} + 2$ is stored in Value.
3. Terminal reads $\text{Value} + 2$.
4. $(\text{Value} + 2) \times 5$ is stored in Value.

Outcome 3: $\text{Value} = \text{Value} \times 5$

1. Terminal reads Value.
2. Local variable reads Value.
3. $\text{Value} + 2$ is stored in Value.
4. $\text{Value} \times 5$ is stored in Value.

Outcome 4: $\text{Value} = \text{Value} + 2$

1. Terminal reads Value.
2. Local variable reads Value.
3. $\text{Value} \times 5$ is stored in Value.
4. $\text{Value} + 2$ is stored in Value.

Although this code is considered a race condition, the code generally behaves less randomly than the first race condition example because LabVIEW usually assigns a consistent order to the operations. However, you should avoid situations such as this one because the order and the behavior of the VI can vary. For example, the order could change when running the VI under different conditions or when upgrading the VI to a newer version of LabVIEW. Fortunately, race conditions of this nature are easily remedied by controlling the data flow.

Self-Review: Quiz

1. You should use variables in your VI whenever possible.
 - a. True
 - b. False

2. When controlling resources, which combination of writers and readers reduces the chance of race conditions?
 - a. One writer, one reader
 - b. One writer, multiple readers
 - c. Multiple writers, one reader
 - d. Multiple writers, multiple readers

Self-Review: Quiz Answers

1. You should use variables in your VI whenever possible.
 - a. True
 - b. False**
You should use variables only when necessary. Use wires to transfer data whenever possible.

2. When controlling resources, which combination of writers and readers reduces the chance of race conditions?
 - a. One writer, one reader**
 - b. One writer, multiple readers**
 - c. Multiple writers, one reader
 - d. Multiple writers, multiple readers

Notes

Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW resources.

National Instruments Technical Support Options

Log in to your National Instruments ni.com User Profile to get personalized access to your services. Visit the following sections of ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
 - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
 - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at ni.com/self-paced-training. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. The NI Alliance Partners joins system integrators, consultants, and hardware vendors to provide comprehensive service and expertise to customers. The program ensures qualified, specialized assistance for application and system development. To learn more, call your local NI office or visit ni.com/alliance.

You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Other National Instruments Training Courses

National Instruments offers several training courses for LabVIEW users. These courses continue the training you received here and expand it to other areas. Visit ni.com/training to purchase course materials or sign up for instructor-led, hands-on courses at locations around the world.

National Instruments Certification

Earning an NI certification acknowledges your expertise in working with NI products and technologies. The measurement and automation industry, your employer, clients, and peers recognize your NI certification credential as a symbol of the skills and knowledge you have gained through experience. Visit ni.com/training for more information about the NI certification program.

LabVIEW Resources

This section describes how you can receive more information regarding LabVIEW.

NI Community

NI Community is an on-line community that unites users from around the world to discuss, share, and learn about LabVIEW. Visit the NI Community to ask questions, get ideas, and share code for LabVIEW projects. Go to ni.com/community for more information.

LabVIEW Books

Many books have been written about LabVIEW programming and applications. The National Instruments Web site contains a list of all the LabVIEW books and links to places to purchase these books. Visit <http://zone.ni.com/devzone/cda/tut/p/id/5389> for more information.

Glossary

A

automatic scaling Ability of scales to adjust to the range of plotted values. On graph scales, autoscaling determines maximum and minimum scale values.

B

block diagram Pictorial description or representation of a program or algorithm. The block diagram consists of executable icons called nodes and wires that carry data between the nodes. The block diagram is the source code for the VI. The block diagram resides in the block diagram window of the VI.

Boolean controls and indicators Front panel objects to manipulate and display Boolean (TRUE or FALSE) data.

broken **Run** button Button that replaces the **Run** button when a VI cannot run because of errors.

broken VI VI that cannot run because of errors; signified by a broken arrow in the broken **Run** button.

C

channel	<p>1. Physical—A terminal or pin at which you can measure or generate an analog or digital signal. A single physical channel can include more than one terminal, as in the case of a differential analog input channel or a digital port of eight lines. A counter also can be a physical channel, although the counter name is not the name of the terminal where the counter measures or generates the digital signal.</p> <p>2. Virtual—A collection of property settings that can include a name, a physical channel, input terminal connections, the type of measurement or generation, and scaling information. You can define NI-DAQmx virtual channels outside a task (global) or inside a task (local). Configuring virtual channels is optional in Traditional NI-DAQ (Legacy) and earlier versions, but is integral to every measurement you take in NI-DAQmx. In Traditional NI-DAQ (Legacy), you configure virtual channels in MAX. In NI-DAQmx, you can configure virtual channels either in MAX or in your program, and you can configure channels as part of a task or separately.</p> <p>3. Switch—A switch channel represents any connection point on a switch. It can be made up of one or more signal wires (commonly one, two, or four), depending on the switch topology. A virtual channel cannot be created with a switch channel. Switch channels may be used only in the NI-DAQmx Switch functions and VIs.</p>
checkbox	Small square box in a dialog box which you can select or clear. Checkboxes generally are associated with multiple options that you can set. You can select more than one checkbox.
conditional terminal	Terminal of a While Loop that contains a Boolean value that determines if the VI performs another iteration.
Context Help window	Window that displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, constants, structures, palettes, properties, methods, events, and dialog box components.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically, such as a knob, push button, or dial.
Controls palette	Palette that contains front panel controls, indicators, and decorative objects.
current VI	VI whose front panel, block diagram, or Icon Editor is the active window.

D

DAQ	<i>See</i> data acquisition (DAQ).
DAQ Assistant	A graphical interface for configuring measurement tasks, channels, and scales.
DAQ device	A device that acquires or generates data and can contain multiple channels and conversion devices. DAQ devices include plug-in devices, PCMCIA cards, and DAQPad devices, which connect to a computer USB or IEEE 1394 port. SCXI modules are considered DAQ devices.
data acquisition (DAQ)	<ol style="list-style-type: none"> 1. Acquiring and measuring analog or digital electrical signals from sensors, acquisition transducers, and test probes or fixtures. 2. Generating analog or digital electrical signals.
data flow	Programming system that consists of executable nodes that execute only when they receive all required input data. The nodes produce output data automatically when they execute. LabVIEW is a dataflow system. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.
data type	Format for information. In LabVIEW, acceptable data types for most VIs and functions are numeric, array, string, Boolean, path, refnum, enumeration, waveform, and cluster.
default	Preset value. Many VI inputs use a default value if you do not specify a value.
device	An instrument or controller you can access as a single entity that controls or monitors real-world I/O points. A device often is connected to a host computer through some type of communication network. <i>See also</i> DAQ device and measurement device.
drag	To use the cursor on the screen to select, move, copy, or delete objects.
driver	Software that controls a specific hardware device, such as a DAQ device.
dynamic data type	Data type used by Express VIs that includes the data associated with a signal and attributes that provide information about the signal, such as the name of the signal or the date and time LabVIEW acquired the data. Attributes specify how the signal appears on a graph or chart.

E

Error list window	Window that displays errors and warnings occurring in a VI and in some cases recommends how to correct the errors.
error message	Indication of a software or hardware malfunction or of an unacceptable data entry attempt.
Express VI	A subVI designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.

F

For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to text-based code: <code>For i = 0 to n - 1, do...</code>
front panel	Interactive user interface of a VI. Front panel appearance imitates physical instruments, such as oscilloscopes and multimeters.
function	Built-in execution element, comparable to an operator, function, or statement in a text-based programming language.
Functions palette	Palette that contains VIs, functions, block diagram structures, and constants.

G

General Purpose Interface Bus	GPIB. Synonymous with HP-IB. The standard bus used for controlling electronic instruments with a computer. Also called IEEE 488 bus because it is defined by ANSI/IEEE Standards 488-1978, 488.1-1987, and 488.2-1992.
graph	2D display of one or more plots. A graph receives and plots data as a block.

I

I/O	Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.
icon	Graphical representation of a node on a block diagram.
indicator	Front panel object that displays output, such as a graph or LED.
instrument driver	A set of high-level functions that control and communicate with instrument hardware in a system.

Instrument I/O Assistant Add-on launched from the Instrument I/O Assistant Express VI that communicates with message-based instruments and graphically parses the response.

L

label Text object used to name or describe objects or regions on the front panel or block diagram.

LabVIEW Laboratory Virtual Instrument Engineering Workbench. LabVIEW is a graphical programming language that uses icons instead of lines of text to create programs.

LED Light-emitting diode.

legend Object a graph or chart owns to display the names and plot styles of plots on that graph or chart.

M

MAX *See* Measurement & Automation Explorer.

Measurement & Automation Explorer The standard National Instruments hardware configuration and diagnostic environment for Windows.

measurement device DAQ devices such as the E Series multifunction I/O (MIO) devices, SCXI signal conditioning modules, and switch modules.

menu bar Horizontal bar that lists the names of the main menus of an application. The menu bar appears below the title bar of a window. Each application has a menu bar that is distinct for that application, although some menus and commands are common to many applications.

N

NI-DAQ Driver software included with all NI DAQ devices and signal conditioning components. NI-DAQ is an extensive library of VIs and ANSI C functions you can call from an application development environment (ADE), such as LabVIEW, to program an NI measurement device, such as the M Series multifunction I/O (MIO) DAQ devices, signal conditioning modules, and switch modules.

NI-DAQmx	The latest NI-DAQ driver with new VIs, functions, and development tools for controlling measurement devices. The advantages of NI-DAQmx over earlier versions of NI-DAQ include the DAQ Assistant for configuring channels and measurement tasks for your device for use in LabVIEW, LabWindows™/CVI™, and Measurement Studio; NI-DAQmx simulation for most supported devices for testing and modifying applications without plugging in hardware; and a simpler, more intuitive API for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.
node	Program execution element. Nodes are analogous to statements, operators, functions, and subroutines in text-based programming languages. On a block diagram, nodes include functions, structures, and subVIs.
numeric controls and indicators	Front panel objects to manipulate and display numeric data.
O	
object	Generic term for any item on the front panel or block diagram, including controls, indicators, structures, nodes, wires, and imported pictures.
Operating tool	Tool to enter data into controls or to operate them.
P	
palette	Displays objects or tools you can use to build the front panel or block diagram.
plot	Graphical representation of an array of data shown either on a graph or a chart.
Positioning tool	Tool to move and resize objects.
project	A collection of LabVIEW files and files not specific to LabVIEW that you can use to create build specifications and deploy or download files to targets.
Project Explorer window	Window in which you can create and edit LabVIEW projects.
Properties dialog boxes	Dialog boxes accessed from the shortcut menu of a control or indicator that you can use to configure how the control or indicator appears in the front panel window.

pull-down menus	Menus accessed from a menu bar. Pull-down menu items are usually general in nature.
PXI	PCI eXtensions for Instrumentation. A modular, computer-based instrumentation platform.
S	
sample	Single analog or digital input or output data point.
scale	Part of graph, chart, and some numeric controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
shortcut menu	Menu accessed by right-clicking an object. Menu items pertain to that object specifically.
string	Representation of a value as text.
structure	Program control element, such as a Flat Sequence structure, Stacked Sequence structure, Case structure, For Loop, While Loop, or Timed Loop.
subpalette	Palette that you access from another palette that is above the subpalette in hierarchy.
subVI	VI used on the block diagram of another VI. Comparable to a subroutine.
T	
task	A collection of one or more channels, timing, triggering, and other properties in NI-DAQmx. A task represents a measurement or generation you want to perform.
template VI	VI that contains common controls and indicators from which you can build multiple VIs that perform similar functions. Access template VIs from the New dialog box.
terminal	Object or region on a node through which data pass.
tip strip	Small yellow text banners that identify the terminal name and make it easier to identify terminals for wiring.
tool	Special cursor to perform specific operations.
toolbar	Bar that contains command buttons to run and debug VIs.

Traditional NI-DAQ (Legacy) An older driver with outdated APIs for developing data acquisition, instrumentation, and control applications for older National Instruments DAQ devices. You should use Traditional NI-DAQ (Legacy) only in certain circumstances. Refer to the *NI-DAQ Readme* for more information about when to use Traditional NI-DAQ (Legacy), including a complete list of supported devices, operating systems, and application software and language versions.

V

VI *See* virtual instrument (VI).

virtual instrument (VI) Program in LabVIEW that models the appearance and function of a physical instrument.

W

waveform Multiple voltage readings taken at a specific sampling rate.

waveform chart Indicator that plots data points at a certain rate.

While Loop Loop structure that repeats a section of code until a condition occurs.

wire Data path between nodes.

Wiring tool Tool to define data paths between terminals.