LabVIEW™

User Manual

April 2003 Edition
Part Number 320999E-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 61 2 9672 8846, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530,
China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00,
Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427,
Hong Kong 2645 3186, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091,
Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9059 6711, Mexico 001 800 010 0793,
Netherlands 31 0 348 433 466, New Zealand 64 09 914 0488, Norway 47 0 32 27 73 00,
Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6 226 5886,
Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00,
Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Patents

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

Chapter 4
Building the Front Panel

Chapter 5
Building the Block Diagram

Chapter 6
Running and Debugging VIs

Chapter 7
Creating VIs and SubVIs

Part II
Building and Editing VIs

Chapter 8
Loops and Structures

Chapter 9
Event-Driven Programming

Chapter 10
Grouping Data Using Strings, Arrays, and Clusters

Chapter 11
Local and Global Variables

Chapter 12
Graphs and Charts

Chapter 13
Graphics and Sound VIs

Chapter 14
File I/O

Chapter 15
Documenting and Printing VIs

Chapter 16
Customizing VIs

Chapter 17
Programmatically Controlling VIs

Chapter 18
Networking in LabVIEW

Chapter 19
Windows Connectivity

Chapter 20
Calling Code from Text-Based Programming Languages

Chapter 21
Formulas and Equations

Appendix A
Organization of LabVIEW

Appendix B
Polymorphic Functions

Appendix C
Comparison Functions

About This Manual

This manual describes the LabVIEW graphical programming environment and techniques for building applications in LabVIEW, such as test and measurement, data acquisition, instrument control, datalogging, measurement analysis, and report generation applications.

Use this manual to learn about LabVIEW programming features, including the LabVIEW user interface and programming workspaces, and the LabVIEW palettes and tools. This manual does not include specific information about each palette, tool, menu, dialog box, control, or built-in VI or function. Refer to the *LabVIEW Help* for more information about these items and for detailed, step-by-step instructions for using LabVIEW features and for building specific applications. Refer to the *LabVIEW Documentation Resources* section of Chapter 1, *Introduction to LabVIEW*, for more information about the *LabVIEW Help* and accessing it.

The *LabVIEW User Manual* is also available in Portable Document Format (PDF). If you select the **Complete** install option, LabVIEW installs PDF versions of all LabVIEW manuals, which you can access by selecting **Help»Search the LabVIEW Bookshelf** in LabVIEW.

Note You must have Adobe Acrobat Reader with Search and Accessibility 5.0.5 or later installed to view the PDFs. Refer to the Adobe Systems Incorporated Web site at `www.adobe.com` to download Acrobat Reader.

You can access the PDFs from the *LabVIEW Help*, but you must install the PDFs to do so. Refer to the *LabVIEW Documentation Resources* section of Chapter 1, *Introduction to LabVIEW*, for information about accessing the PDFs in the *LabVIEW Bookshelf*.

Organization of This Manual

The *LabVIEW User Manual* includes two sections. Part I, *LabVIEW Concepts*, describes programming concepts for building applications in LabVIEW. The chapters in this section introduce you to the LabVIEW programming environment and help you plan your application.

Part II, *Building and Editing VIs*, describes LabVIEW features, VIs, and functions you can use to make your applications operate in specific ways. The chapters in this section describe the use of each LabVIEW feature and outline each class of VIs and functions.

Conventions

The following conventions appear in this manual:

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

Part I

LabVIEW Concepts

This part describes programming concepts for building applications in LabVIEW. The chapters in this section introduce you to the LabVIEW programming environment and help you plan your application.

Part I, *LabVIEW Concepts*, contains the following chapters:

- Chapter 1, *Introduction to LabVIEW*, describes LabVIEW, its extensive documentation, and tools to help you design and build VIs.
- Chapter 2, *Introduction to Virtual Instruments*, describes the components of virtual instruments, or VIs.
- Chapter 3, *LabVIEW Environment*, describes the LabVIEW palettes, tools, and menus you use to build the front panels and block diagrams of VIs. This chapter also describes how to customize the LabVIEW palettes and set several work environment options.
- Chapter 4, *Building the Front Panel*, describes how to build the front panel of a VI.
- Chapter 5, *Building the Block Diagram*, describes how to build the block diagram of a VI.
- Chapter 6, *Running and Debugging VIs*, describes how to configure how a VI runs and to identify problems with block diagram organization or with the data passing through the block diagram.
- Chapter 7, *Creating VIs and SubVIs*, describes how to create your own VIs and subVIs, distribute VIs, and build stand-alone applications and shared libraries.

1

Introduction to LabVIEW

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

In LabVIEW, you build a user interface with a set of tools and objects. The user interface is known as the front panel. You then add code using graphical representations of functions to control the front panel objects. The block diagram contains this code. In some ways, the block diagram resembles a flowchart.

You can purchase several add-on software toolsets for developing specialized applications. All the toolsets integrate seamlessly in LabVIEW. Refer to the National Instruments Web site at `ni.com` for more information about these toolsets.

LabVIEW Documentation Resources

LabVIEW includes extensive documentation for new and experienced LabVIEW users. All LabVIEW manuals and Application Notes are also available as PDFs. You must have Adobe Acrobat Reader with Search and Accessibility 5.0.5 or later installed to view the PDFs. Refer to the Adobe Systems Incorporated Web site at `www.adobe.com` to download Acrobat Reader. Refer to the National Instruments Product Manuals Library at `ni.com/manuals` for updated documentation resources.

- **LabVIEW Bookshelf**—Use this PDF to search PDF versions of all the LabVIEW manuals and Application Notes. Access the *LabVIEW Bookshelf* by selecting **Help»Search the LabVIEW Bookshelf**.
- **Getting Started with LabVIEW**—Use this manual to familiarize yourself with the LabVIEW graphical programming environment and the basic LabVIEW features you use to build data acquisition and instrument control applications.
- **LabVIEW Quick Reference Card**—Use this card as a reference for information about help resources, keyboard shortcuts, terminal data types, and tools for editing, execution, and debugging.

- **LabVIEW User Manual**—Use this manual to learn about LabVIEW 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.
- **LabVIEW Help**—Use this help file as a reference for information about LabVIEW palettes, menus, tools, VIs, and functions. The *LabVIEW Help* also includes step-by-step instructions for using LabVIEW features. Access the *LabVIEW Help* by selecting **Help» VI, Function, and How-To Help**.

  The *LabVIEW Help* includes links to the following resources:

  – *LabVIEW Bookshelf*, which includes PDF versions of all the LabVIEW manuals and Application Notes

  – Technical support resources on the National Instruments Web site, such as the NI Developer Zone, the KnowledgeBase, and the Product Manuals Library

Note (Mac OS and UNIX) National Instruments recommends that you use Netscape 6.0 or later or Internet Explorer 5.0 or later to view the *LabVIEW Help*.

- **LabVIEW Measurements Manual**—Use this manual to learn more about building data acquisition and instrument control applications in LabVIEW. If you are a new LabVIEW user, read the *Getting Started with LabVIEW* manual and the *LabVIEW User Manual* before you read this manual.
- **LabVIEW Application Builder User Guide**—Use this document to learn about the LabVIEW Application Builder, which is included in the LabVIEW Professional Development System and is available for purchase separately. This user guide contains instructions for installing the Application Builder, describes system requirements for applications, and describes the changes introduced between previous versions and the current version. This user guide also describes caveats and recommendations to consider when you build a VI into an application or shared library.
- **LabVIEW Development Guidelines**—Use this manual to learn how to build VIs that are easy to understand, use, and revise. This manual describes project tracking, design, and documentation techniques. This manual also contains recommended style guidelines.

Note The printed *LabVIEW Development Guidelines* manual is available only in the LabVIEW Professional Development System. The PDF is available in all packages of LabVIEW.

- **LabVIEW Analysis Concepts**—Use this manual to learn about the analysis concepts used by LabVIEW. This manual includes information about signal generation, the fast Fourier transform (FFT) and the discrete Fourier transform (DFT), smoothing windows, curve fitting, linear algebra, fundamental concepts of probability and statistics, and point-by-point analysis for real-time analysis.

Note The *LabVIEW Analysis Concepts* manual is available only as a PDF.

- **Using External Code in LabVIEW**—Use this manual to learn how to use Code Interface Nodes and external subroutines to import code written in text-based programming languages. The manual includes information about calling DLLs, shared external subroutines, libraries of functions, memory and file manipulation routines, and diagnostic routines.

Note The *Using External Code in LabVIEW* manual is available only as a PDF.

- **LabVIEW Release Notes**—Use these release notes to install and uninstall LabVIEW. The release notes also describe the system requirements for the LabVIEW software and known issues with LabVIEW.
- **LabVIEW Upgrade Notes**—Use these upgrade notes to upgrade LabVIEW for Windows, Mac OS, and UNIX to the latest version. The upgrade notes also describe new features and issues you might encounter when you upgrade.
- **LabVIEW Application Notes**—Use the LabVIEW Application Notes to learn about advanced or specialized LabVIEW concepts and applications. Refer to the NI Developer Zone at `ni.com/zone` for new and updated Application Notes.
- **LabVIEW VXI VI Reference Manual**—Use this manual to learn about the VXI VIs for LabVIEW. This manual is a companion guide to the *NI-VXI Programmer Reference Manual* that comes with the VXI hardware. National Instruments recommends using VISA technology for configuring, programming, and troubleshooting instrumentation systems consisting of VXI hardware.

Note The *LabVIEW VXI VI Reference Manual* is available only as a PDF.

LabVIEW Template VIs, Example VIs, and Tools

Use the LabVIEW template VIs, example VIs, and tools to help you design and build VIs.

LabVIEW Template VIs

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

LabVIEW Example VIs

LabVIEW includes hundreds of example VIs you can use and incorporate into your own VIs. You can modify an example to fit your application, or you can copy and paste from one or more examples into your own VI. Browse or search the example VIs by selecting **Help»Find Examples**. Refer to the NI Developer Zone at `ni.com/zone` for additional example VIs.

LabVIEW Tools

LabVIEW includes many tools to help you quickly configure your measurement devices. You can access the following tools from the **Tools** menu.

- **(Windows)** Measurement & Automation Explorer (MAX) helps you configure National Instruments hardware and software.
- **(Mac OS 9 or earlier)** The NI-DAQ Configuration Utility helps you configure National Instruments DAQ hardware.
- **(Mac OS 9 or earlier)** The DAQ Channel Wizard helps you define what type of device is connected to the DAQ hardware channels. After you define a channel, the DAQ Channel Wizard remembers the settings.
- **(Mac OS 9 or earlier)** The DAQ Channel Viewer lists the configured DAQ channels.
- **(Mac OS 9 or earlier)** The DAQ Solution Wizard helps you find solutions for common DAQ applications. You can choose from example VIs or create custom VIs.

Use the DAQ Assistant to graphically configure channels or common measurement tasks. You can access the DAQ Assistant in the following ways:

- Place the DAQ Assistant Express VI on the block diagram.
- Right-click a DAQmx Global Channel control and select **New Channel (DAQ Assistant)** from the shortcut menu. Right-click a DAQmx Task Name control and select **New Task (DAQ Assistant)** from the shortcut menu. Right-click a DAQmx Scale Name control and select **New Scale (DAQ Assistant)** from the shortcut menu.
- Launch Measurement & Automation Explorer and select **Data Neighborhood** or **Scales** from the **Configuration** tree. Click the **Create New** button. Configure a NI-DAQmx channel, task, or scale.

Refer to the *LabVIEW Measurements Manual* for more information about using the DAQ Assistant.

2

Introduction to Virtual Instruments

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

A VI contains the following three components:

- **Front panel**—Serves as the user interface.
- **Block diagram**—Contains the graphical source code that defines the functionality of the VI.
- **Icon and connector pane**—Identifies the VI so that you can use the VI in another VI. A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages.

For more information…

Refer to the *LabVIEW Help* for more information about creating VIs and subVIs.

Front Panel

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

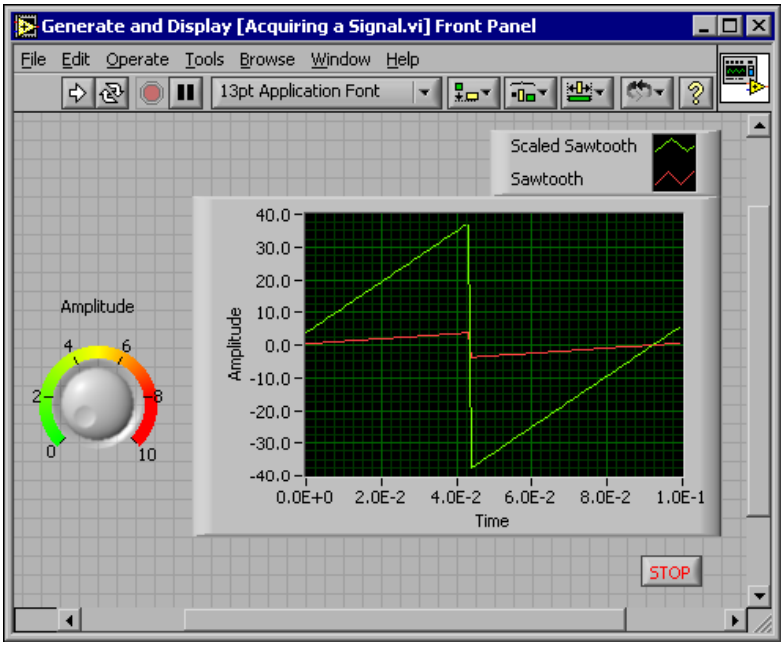


Figure 2-1. Example of a Front Panel

You build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays. 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. Refer to Chapter 4, *Building the Front Panel*, for more information about the front panel.

Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code. Front panel objects appear as terminals on the block diagram. Refer to Chapter 5, *Building the Block Diagram*, for more information about the block diagram.

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

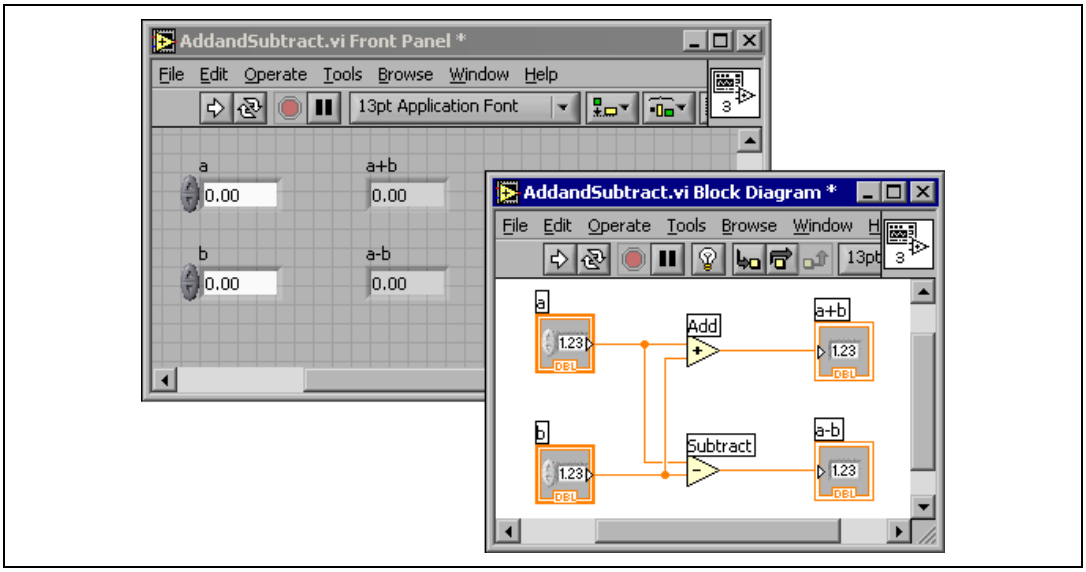


Figure 2-2. Example of a Block Diagram and Corresponding Front Panel

Terminals

The terminals represent the data type of the control or indicator. You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a knob icon terminal shown at left, represents a knob on the front panel. The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric. A DBL terminal, shown at left, represents a double-precision, floating-point numeric control or indicator. Refer to the *Control and Indicator Data Types* section of Chapter 5, *Building the Block Diagram*, for more information about data types in LabVIEW and their graphical representations.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data you enter into the front panel controls (**a** and **b** in Figure 2-2) 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 internal calculations, they produce new data values. The data flow to the indicator terminals, where they exit the block diagram, reenter the front panel, and appear in front panel indicators (**a+b** and **a-b** in Figure 2-2).

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. The Add and Subtract functions in Figure 2-2 are nodes. Refer to the *Block Diagram Nodes* section of Chapter 5, *Building the Block Diagram*, for more information about nodes.

Wires

You transfer data among block diagram objects through wires. In Figure 2-2, wires connect the control and indicator terminals to the Add and Subtract functions. 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. Refer to the *Using Wires to Link Block Diagram Objects* section of Chapter 5, *Building the Block Diagram*, for more information about wires.

Structures

Structures are graphical representations of the loops and case statements of text-based programming languages. Use structures on the block diagram to repeat blocks of code and to execute code conditionally or in a specific order. Refer to Chapter 8, *Loops and Structures*, for examples and more information about structures.

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. Every VI displays an icon, such as the one shown at left, in the upper right corner of the front panel and block diagram windows. An 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. You can double-click the icon to customize or edit it. Refer to the *Creating an Icon* section of Chapter 7, *Creating VIs and SubVIs*, for more information about icons.

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

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

When you view the connector pane for the first time, you see a connector pattern. You can select a different pattern if you want to. The connector pane generally has one terminal for each control or indicator on the front panel. You can assign up to 28 terminals to a connector pane. If you anticipate changes to the VI that would require a new input or output, leave extra terminals unassigned. Refer to the *Setting up the Connector Pane* section of Chapter 7, *Creating VIs and SubVIs*, for more information about setting up connector panes.

Note Try not to assign more than 16 terminals to a VI. Too many terminals can reduce the readability and usability of the VI.

Using and Customizing VIs and SubVIs

After you build a VI and create its icon and connector pane, you can use it as a subVI. Refer to the *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about subVIs.

You can save VIs as individual files, or you can group several VIs together and save them in a VI library. Refer to the *Saving VIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about saving VIs in libraries.

You can customize the appearance and behavior of a VI. You also can create custom menus for every VI you build, and you can configure VIs to show or hide menu bars. Refer to Chapter 16, *Customizing VIs*, for more information about customizing a VI.

3

LabVIEW Environment

Use the LabVIEW palettes, tools, and menus to build the front panels and block diagrams of VIs. You can customize the **Controls** and **Functions** palettes, and you can set several work environment options.

For more information…

Refer to the *LabVIEW Help* for more information about using the palettes, menus, and toolbar, and customizing your work environment.

Controls Palette

The **Controls** palette is available only on the front panel. The **Controls** palette contains the controls and indicators you use to create the front panel. The controls and indicators are located on subpalettes based on the types of controls and indicators. Refer to the *Front Panel Controls and Indicators* section of Chapter 4, *Building the Front Panel*, for more information about the types of controls and indicators. The controls and indicators located on the **Controls** palette depend on the palette view currently selected. Refer to the *Creating and Editing Custom Palette View* section of this chapter for more information about palette views.

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

You can change the way the **Controls** palette appears. Refer to the *Customizing the Controls and Functions Palettes* section of this chapter for more information about customizing the **Controls** palette.

Functions Palette

The **Functions** palette is available only on the block diagram. The **Functions** palette contains the VIs and functions you use to build the block

diagram. The VIs and functions are located on subpalettes based on the types of VIs and functions. Refer to the *Functions Overview* section of Chapter 5, *Building the Block Diagram*, for more information about the types of VIs and functions. The VIs and functions located on the **Functions** palette depend on the palette view currently selected. Refer to the *Creating and Editing Custom Palette View* section of this chapter for more information about palette views.

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

You can change the way the **Functions** palette appears. Refer to the *Customizing the Controls and Functions Palettes* section of this chapter for more information about customizing the **Functions** palette.

Navigating the Controls and Functions Palettes

When you click a subpalette icon, the entire palette changes to the subpalette you selected. Click an object on the palette to place the object on the cursor so you can place it on the front panel or block diagram. You also can right-click a VI icon on the palette and select **Open VI** from the shortcut menu to open the VI.

Use the following buttons on the **Controls** and **Functions** palette toolbars to navigate the palettes, to configure the palettes, and to search for controls, VIs, and functions:

- **Up**—Takes you up one level in the palette hierarchy. Click this button and hold the mouse button down to display a shortcut menu that lists each subpalette in the path to the current subpalette. Select a subpalette name in the shortcut menu to navigate to the subpalette.

- **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 to Palette** button to exit search mode and return to the palette.

- **Options**—Displays the **Controls/Functions Palettes** page of the **Options** dialog box, in which you can select a palette view and a format for the palettes.

- **Restore Palette Size**—Resizes the palette to its default size. This button appears only if you resize the **Controls** or **Functions** palette.

Tools Palette

The **Tools** palette is available on the front panel and the block diagram. A tool is a special operating mode of the mouse cursor. The cursor corresponds to the icon of the tool selected in the palette. Use the tools to operate and modify front panel and block diagram objects.

Select **Window»Show Tools Palette** to display the **Tools** palette. You can place the **Tools** palette anywhere on the screen. LabVIEW retains the **Tools** palette position so when you restart LabVIEW, the palette appears in the same position.

Tip Press the <Shift> key and right-click to display a temporary version of the **Tools** palette at the location of the cursor.

If automatic tool selection is enabled and you move the cursor over objects on the front panel or block diagram, LabVIEW automatically selects the corresponding tool from the **Tools** palette. You can disable automatic tool selection by clicking the **Automatic Tool Selection** button on the **Tools** palette, shown at left. Press the <Shift-Tab> keys or click the **Automatic Tool Selection** button to enable automatic tool selection again. You also can disable automatic tool selection by manually selecting a tool on the **Tools** palette. Press the <Tab> key or click the **Automatic Tool Selection** button on the **Tools** palette to enable automatic tool selection again.

Menus and the Toolbar

Use the menu and toolbar items to operate and modify front panel and block diagram objects. Use the toolbar buttons to run VIs.

Menus

The menus at the top of a VI window contain items common to other applications, such as **Open**, **Save**, **Copy**, and **Paste**, and other items specific to LabVIEW. Some menu items also list shortcut key combinations.

(Mac OS) The menus appear at the top of the screen.

(Windows and UNIX) The menus display only the most recently used items by default. Click the arrows at the bottom of a menu to display all items.You can display all menu items by default by selecting **Tools»Options** and selecting **Miscellaneous** from the top pull-down menu.

Note Some menu items are unavailable while a VI is in run mode.

Shortcut Menus

The most often-used menu is the object shortcut menu. All LabVIEW objects and empty space on the front panel and block diagram have associated shortcut menus. Use the shortcut menu items to change the look or behavior of front panel and block diagram objects. To access the shortcut menu, right-click the object, front panel, or block diagram.

Shortcut Menus in Run Mode

When a VI is running, or is in run mode, all front panel objects have an abridged set of shortcut menu items. Use the abridged shortcut menu items to cut, copy, or paste the contents of the object, to set the object to its default value, or to read the description of the object.

Some of the more complex controls have additional options. For example, the array shortcut menu includes items to copy a range of values or go to the last element of the array.

Toolbar

Use the toolbar buttons to run and edit a VI. When you run a VI, buttons appear on the toolbar that you can use to debug the VI.

Context Help Window

The **Context Help** window displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, constants, structures, palettes, properties, methods, events, and dialog box components. You also can use the **Context Help** window to determine exactly where to connect wires to a VI or function. Refer to the *Manually Wiring Objects* section of Chapter 5, *Building the Block Diagram*, for more information about using the **Context Help** window to wire objects.

Select **Help»Show Context Help** to display the **Context Help** window. You also can display the **Context Help** window by clicking the **Show Context Help Window** button on the toolbar, shown at left, or by pressing the <Ctrl-H> keys. **(Mac OS)** Press the <Command-H> keys. **(UNIX)** Press the <Alt-H> keys.

You can place the **Context Help** window anywhere on the screen. The **Context Help** window resizes to accommodate each object description. You also can resize the **Context Help** window to set its maximum size. LabVIEW retains the **Context Help** window position and size so when you restart LabVIEW, the window appears in the same position and has the same maximum size.

You can lock the current contents of the **Context Help** window so the contents of the window do not change when you move the cursor over different objects. Select **Help»Lock Context Help** to lock or unlock the current contents of the **Context Help** window. You also can lock or unlock the contents of the window by clicking the **Lock** button in the **Context Help** window, shown at left, or by pressing the <Ctrl-Shift-L> keys. **(Mac OS)** Press the <Command-Shift-L> keys. **(UNIX)** Press the <Alt-Shift-L> keys.

Click the **Show Optional Terminals and Full Path** button in the **Context Help** window, shown at left, to display the optional terminals of a connector pane and to display the full path to a VI. Refer to the *Setting Required, Recommended, and Optional Inputs and Outputs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about optional terminals.

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

Refer to the *Creating VI and Object Descriptions* section of Chapter 15, *Documenting and Printing VIs*, for information about creating descriptions to display in the **Context Help** window.

Customizing Your Work Environment

You can customize the Controls and Functions palettes, and you can use the Options dialog box to select a palette view and set other work environment options.

Customizing the Controls and Functions Palettes

You can customize the **Controls** and **Functions** palettes in the following ways:

- Add VIs and controls to the palettes.
- Set up different views for different users, hiding some VIs and functions to make LabVIEW easier to use for one user while providing the full palettes for another user.
- Rearrange the built-in palettes to make the VIs and functions you use frequently more accessible.
- Convert a set of ActiveX controls into custom controls and add them to the palettes.
- Add toolsets to the palettes.

Caution Do *not* save your own VIs and controls in the `vi.lib` directory because LabVIEW overwrites these files when you upgrade or reinstall. Save your VIs and controls in the `user.lib` directory to add them to the **Functions** and **Controls** palettes.

Adding VIs and Controls to the User and Instrument Drivers Subpalettes

The simplest method for adding VIs and controls to the **Functions** and **Controls** palettes is to save them in the `labview\user.lib` directory. When you restart LabVIEW, the **User Libraries** and **User Controls** palettes contain subpalettes for each directory, VI library (`.llb`), or menu (`.mnu`) file in `labview\user.lib` and icons for each file in `labview\user.lib`. After you add files to or remove files from specific directories, LabVIEW automatically updates the palettes when you restart LabVIEW.

The **Instrument Drivers** palette corresponds to the `labview\instr.lib` directory. Save instrument drivers in this directory to add them to the **Functions** palette.

When you add VIs or controls to the **Functions** and **Controls** palettes using this method, you cannot set the name of each subpalette or the exact location of the VIs or controls on the palettes.

Creating and Editing Custom Palette View

To control the name of each subpalette and the exact location of the controls and VIs you add to the **Controls** and **Functions** palettes, you must create a custom palette view. LabVIEW includes two built-in palette

views—Express and Advanced. Select **Tools»Advanced»Edit Palette Views** to create or edit custom palette views.

Note You cannot edit a built-in palette view.

LabVIEW stores **Controls** and **Functions** palette information in the `labview\menus` directory. The `menus` directory contains directories that correspond to each view that you create or install. If you run LabVIEW on a network, you can define individual menus directories for each user, which makes it easy to transfer views to other users.

When you create a new view of a palette, LabVIEW uses a copy of the original built-in view, upon which you can apply any changes. LabVIEW copies the original built-in palette located in the `labview\menus` directory before you make any changes. The protection of the built-in palettes ensures that you can experiment with the palettes without corrupting the original view.

How LabVIEW Stores Views

The `.mnu` files and `.llb` files contain one **Controls** palette and one **Functions** palette each. In addition, each file contains an icon for the **Controls** and **Functions** palettes. You must store each subpalette you create in a separate `.mnu` file.

When you select a view, LabVIEW checks the `menus` directory for a directory that corresponds to that view. It builds the top-level **Controls** and **Functions** palettes and subpalettes from the `root.mnu` file in that directory that LabVIEW automatically creates every time you create a view.

For each VI or control, LabVIEW creates an icon on the palette. For each subdirectory, `.mnu` file, or `.llb` file, LabVIEW creates a subpalette on the palette.

Building ActiveX Subpalettes

If you use ActiveX controls on the front panel, select **Tools»Advanced»Import ActiveX Controls** to convert a set of ActiveX controls to custom controls and add them to the **Controls** palette. LabVIEW saves the controls in the `user.lib` directory by default because all files and directories in `user.lib` automatically appear in the palettes.

Representing Toolsets and Modules in the Palettes

Toolsets and modules with controls or VIs in `vi.lib\addons` appear on the **Controls** and **Functions** palettes after you restart LabVIEW. In the built-in Express palette view, toolsets and modules install subpalettes on the **All Controls** and **All Functions** subpalettes. In the built-in Advanced palette view, toolsets and modules install subpalettes on the top level of the **Controls** and **Functions** palettes.

If you installed toolset or module controls and VIs outside the `vi.lib\addons` directory, you can move the controls and VIs to the `vi.lib\addons` directory to add them to the palettes.

Setting Work Environment Options

Select **Tools»Options** to customize LabVIEW. Use the **Options** dialog box to set options for front panels, block diagrams, paths, performance and disk issues, the alignment grid, palettes, undo, debugging tools, colors, fonts, printing, the **History** window, and other LabVIEW features.

Use the top pull-down menu in the **Options** dialog box to select among the different categories of options.

How LabVIEW Stores Options

You do not have to edit options manually or know their exact format because the **Options** dialog box does it for you. LabVIEW stores options differently on each platform.

Windows

LabVIEW stores options in a `labview.ini` file in the LabVIEW directory. The file format is similar to other `.ini` files. It begins with a LabVIEW section marker followed by the option name and the values, such as `offscreenUpdates=True`.

If you want to use a different options file, specify the file in the shortcut you use to start LabVIEW. For example, to use an options file on your computer named `lvrc` instead of `labview.ini`, right-click the LabVIEW icon on the desktop and select **Properties**. Click the **Shortcut** tab and type `labview -pref lvrc` in the **Target** text box.

Mac OS

LabVIEW stores options in the `LabVIEW Preferences` text file in the **System»Preferences** folder.

If you want to use a different options file, copy the `LabVIEW Preferences` file to the **LabVIEW** folder and make options changes in the **Options** dialog box. When you launch LabVIEW, it first looks for an options file in the **LabVIEW** folder. If it does not find the file there, it looks in the **System** folder. If it does not find the file there, it creates a new one in the **System** folder. LabVIEW writes all changes you make in the **Options** dialog box to the first `LabVIEW Preferences` file it finds.

UNIX

LabVIEW stores options in the `.labviewrc` file in your home directory. If you change an option in the **Options** dialog box, LabVIEW writes the change to the `.labviewrc` file. You can create a `labviewrc` file in the program directory to store options that are the same for all users, such as the VI search path. Use the `.labviewrc` file to store options that are different for each user, such as font or color settings, because entries in the `.labviewrc` file in your home directory override conflicting entries in the program directory.

For example, if you installed the LabVIEW files in `/opt/labview`, LabVIEW first reads options from `/opt/labview/labviewrc`. If you change an option in the **Options** dialog box, such as the application font, LabVIEW writes that change to the `.labviewrc` file. The next time you start LabVIEW, it uses the application font option in the `.labviewrc` file instead of the default application font defined in `/opt/labview/labviewrc`.

Option entries consist of an option name followed by a colon and a value. The option name is the executable followed by a period (`.`) and an option. When LabVIEW searches for option names, the search is case sensitive. You can enclose the option value in double or single quotation marks. For example, to use a default precision of double, add the following entry to the `.labviewrc` file in your home directory.

```
labview.defPrecision : double
```

If you want to use a different options file, specify the file on the command line when you start LabVIEW. For example, to use a file named `lvrc` in the `test` directory instead of `.labviewrc`, type `labview -pref /test/lvrc`. LabVIEW writes all changes you make in the **Options** dialog box to the `lvrc` options file. When you specify an options file on the command line, LabVIEW still reads the `labviewrc` file in the program directory, but the options file specified on the command line overrides conflicting entries in the program directory.

4

Building the Front Panel

The front panel is the user interface of a VI. Generally, you design the front panel first, then design the block diagram to perform tasks on the inputs and outputs you create on the front panel. Refer to Chapter 5, *Building the Block Diagram*, for more information about the block diagram.

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

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

For more information…

Refer to the *LabVIEW Help* for more information about designing and configuring the front panel.

Configuring Front Panel Objects

Use property dialog boxes or shortcut menus to configure how controls and indicators appear or behave on the front panel. Use property dialog boxes when you want to configure a front panel control or indicator through a dialog box that includes context help or when you want to set several properties at once for an object. Use shortcut menus to quickly configure common control and indicator properties. Options in the property dialog boxes and shortcut menus differ depending on the front panel object. Any option you set using a shortcut menu overrides the option you set with a property dialog box. Refer to the *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note for more information about creating and using custom controls, indicators, and type definitions.

Right-click a control or indicator on the front panel and select **Properties** from the shortcut menu to access the property dialog box for that object. You cannot access property dialog boxes for a control or indicator while a VI runs.

Showing and Hiding Optional Elements

Front panel controls and indicators have optional elements you can show or hide. Set the visible elements for the control or indicator on the **Appearance** tab of the property dialog box for the front panel object. You also can set the visible elements by right-clicking an object and selecting **Visible Items** from the shortcut menu. Most objects have a label and a caption. Refer to the *Labeling* section of this chapter for more information about labels and captions.

Changing Controls to Indicators and Indicators to Controls

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

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

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

Replacing Front Panel Objects

You can replace a front panel object with a different control or indicator. When you right-click an object and select **Replace** from the shortcut menu, a temporary **Controls** palette appears, even if the **Controls** palette is already open. Select a control or indicator from the temporary **Controls** palette to replace the current object on the front panel.

Selecting **Replace** from the shortcut menu preserves as much information as possible about the original object, such as its name, description, default data, dataflow direction (control or indicator), color, size, and so on. However, the new object retains its own data type. Wires from the terminal of the object or local variables remain on the block diagram, but they might

be broken. For example, if you replace a numeric terminal with a string terminal, the original wire remains on the block diagram, but is broken.

The more the new object resembles the object you are replacing, the more original characteristics you can preserve. For example, if you replace a slide with a different style slide, the new slide has the same height, scale, value, name, description, and so on. If you replace the slide with a string control instead, LabVIEW preserves only the name, description, and dataflow direction because a slide does not have much in common with a string control.

You also can paste objects from the clipboard to replace existing front panel controls and indicators. This method does not preserve any characteristics of the old object, but the wires remain connected to the object.

Configuring the Front Panel

You can customize the front panel by setting the tabbing order of front panel objects, by using imported graphics, and by setting front panel objects to automatically resize when the window size changes.

Setting Keyboard Shortcuts for Controls

You can assign keyboard shortcuts to controls so users can navigate the front panel without a mouse. Right-click the control and select **Advanced» Key Navigation** from the shortcut menu to display the **Key Navigation** dialog box.

Note LabVIEW does not respond to keyboard shortcuts for hidden controls.

When a user enters the keyboard shortcut while running the VI, the associated control receives the focus. If the control is a text or digital control, LabVIEW highlights the text so you can edit it. If the control is Boolean, press the spacebar or the <Enter> key to change its value.

The **Advanced»Key Navigation** shortcut menu item is dimmed for indicators because you cannot enter data in an indicator.

Note You also can use the Key Down event to generate an event when the user presses a specific key on the keyboard.

Refer to the *Key Navigation* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about setting keyboard shortcuts in a user interface.

Controlling Button Behavior with Key Navigation

You can associate function keys with various buttons that control the behavior of a front panel. You can configure a button in a VI to behave like a dialog box button so that pressing the <Enter> key is the same as clicking the button.

If you associate the <Enter> key with a dialog box button, LabVIEW automatically draws that button with a thick border around it.

If you associate the <Enter> key with a control, the user cannot enter a carriage return in any string control on that front panel. All strings on that front panel are limited to a single line. You can use scrollbars to navigate longer strings.

If you navigate to a Boolean control and press the <Enter> key, the Boolean control changes, even if another control uses the <Enter> key as its keyboard shortcut. The assigned <Enter> keyboard shortcut applies only when a Boolean control is not selected.

Setting the Tabbing Order of Front Panel Objects

Controls and indicators on a front panel have an order, called tabbing order, that is unrelated to their position on the front panel. The first control or indicator you create on the front panel is element `0`, the second is `1`, and so on. If you delete a control or indicator, the tabbing order adjusts automatically.

The tabbing order determines the order in which LabVIEW selects controls and indicators when the user presses the <Tab> key while a VI runs. The tabbing order also determines the order in which the controls and indicators appear in the records of datalog files you create when you log the front panel data. Refer to the *Logging Front Panel Data* section of Chapter 14, *File I/O*, for more information about logging data.

You can set the tabbing order of front panel objects by selecting **Edit»Set Tabbing Order**.

To prevent users from accessing a control by pressing the <Tab> key while a VI runs, place a checkmark in the **Skip this control when tabbing** checkbox in the **Key Navigation** dialog box.

Coloring Objects

You can change the color of many objects but not all of them. For example, block diagram terminals of front panel objects and wires use specific colors for the type and representation of data they carry, so you cannot change them.

Use the Coloring tool to right-click an object or workspace to change the color of front panel objects or the front panel and block diagram workspaces. You also can change the default colors for most objects by selecting **Tools»Options** and selecting **Colors** from the top pull-down menu.

Refer to the *Colors* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about using color to design a user interface.

Using Imported Graphics

You can import graphics from other applications to use as front panel backgrounds, items in ring controls, and parts of other controls and indicators. Refer to the *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note for more information about using graphics in controls.

LabVIEW supports most standard graphic formats, including BMP, JPEG, animated GIF, MNG, animated MNG, and PNG. LabVIEW also supports transparency.

To import a graphic, copy it to the clipboard and paste it on the front panel. You also can select **Edit»Import Picture from File**.

Note (Windows and Mac OS) If you import an image by copying and pasting it, the image loses any transparency.

Refer to the `examples\general\controls\custom.llb` for examples of controls with imported graphics. Refer to the *Graphics and Custom Controls* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about using graphics to design a user interface.

Aligning and Distributing Objects

Select **Operate»Enable Alignment Grid on Panel** to enable the alignment grid on the front panel and align objects as you place them. Select **Operate»Disable Alignment Grid on Panel** to disable the alignment grid and use the visible grid to align objects manually. You also can press the <Ctrl-#> keys to enable or disable the alignment grid. On French keyboards, press the <Ctrl-”> keys.

(Mac OS) Press the <Command-*> keys. **(Sun)** Press the <Meta-#> keys. **(Linux)** Press the <Alt-#> keys.

You also can use the alignment grid on the block diagram.

Select **Tools»Options** and select **Alignment Grid** from the top pull-down menu to hide or customize the grid.

To align objects after you place them, select the objects and select the **Align Objects** pull-down menu on the toolbar. To space objects evenly, select the objects and select the **Distribute Objects** pull-down menu on the toolbar.

Grouping and Locking Objects

Use the Positioning tool to select the front panel objects you want to group and lock together. Click the **Reorder** button on the toolbar and select **Group** or **Lock** from the pull-down menu. Grouped objects maintain their relative arrangement and size when you use the Positioning tool to move and resize them. Locked objects maintain their location on the front panel and you cannot delete them until you unlock them. You can set objects to be grouped and locked at the same time. Tools other than the Positioning tool work normally with grouped or locked objects.

Resizing Objects

You can change the size of most front panel objects. When you move the Positioning tool over a resizable object, resizing handles or circles appear at the points where you can resize the object. When you resize an object, the font size remains the same. Resizing a group of objects resizes all the objects within the group.

Some objects change size only horizontally or vertically when you resize them, such as digital numeric controls and indicators. Others keep the same proportions when you resize them, such as knobs. The Positioning cursor appears the same, but the dashed border that surrounds the object moves in only one direction.

You can manually restrict the growth direction when you resize an object. To restrict the growth vertically or horizontally or to maintain the current proportions of the object, press the <Shift> key while you click and drag the resizing handles or circles. To resize an object around its center point, press the <Ctrl> key while you click and drag the resizing handles or circles.

(Mac OS) Press the <Option> key. **(Sun)** Press the <Meta> key. **(Linux)** Press the <Alt> key.

To resize multiple objects to the same size, select the objects and select the **Resize Objects** pull-down menu on the toolbar.You can resize all the selected objects to the width or height of the largest or smallest object, and you can resize all the selected objects to a specific size in pixels.

Scaling Front Panel Objects

You can set the front panel objects to scale, or automatically resize in relation to the window size, when you resize the front panel window. You can set one object on the front panel to scale, or you can set all objects on the front panel to scale. However, you cannot set multiple objects to scale on the front panel unless you set all of them to scale or unless you group the objects first. To scale an object, select the object and select **Edit»Scale Object with Panel**.

If you set a single front panel object to scale, the object resizes itself automatically in relation to any change in the front panel window size. The other objects on the front panel reposition themselves to remain consistent with their previous placement on the front panel but do not scale to fit the new window size of the front panel.

Immediately after you designate a single object on the front panel to scale automatically, gray lines outline several regions on the front panel, as shown in Figure 4-1. The regions define the positions of the other front panel objects in relation to the object you want to scale. When you resize the front panel window, the object you set to scale automatically resizes and repositions itself relative to its original location. The gray lines disappear when you run the VI.

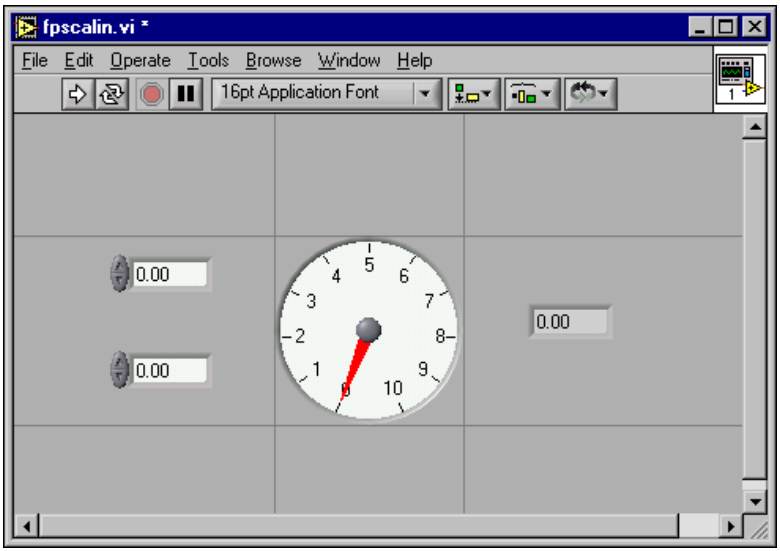


Figure 4-1. Front Panel with Object Set to Scale

When LabVIEW scales objects automatically, it follows the same conventions as when you resize an object manually. For example, some objects can resize only horizontally or vertically, and the font size remains the same when you resize an object.

After LabVIEW automatically scales an object, the object might not scale back to its exact original size when you size the window back to its original position. Before you save the VI, select **Edit»Undo** to restore the original front panel window and object sizes.

You can set an array to scale or set the objects within an array to scale. If you set the array to scale, you adjust the number of rows and columns you can see within the array. If you set the objects within the array to scale, you always see the same number of rows and columns, though different sizes, within the array.

You also can set a cluster to scale or set the objects within the cluster to scale. If you set the objects within the cluster to scale, the cluster adjusts as well.

Adding Space to the Front Panel without Resizing the Window

You can add space to the front panel without resizing the window. To increase the space between crowded or tightly grouped objects, press the <Ctrl> key and use the Positioning tool to click the front panel workspace. While holding the key combination, drag out a region the size you want to insert.

(Mac OS) Press the <Option> key. **(Sun)** Press the <Meta> key. **(Linux)** Press the <Alt> key.

A rectangle marked by a dashed border defines where space will be inserted. Release the key combination to add the space.

Front Panel Controls and Indicators

Use the front panel controls and indicators located on the **Controls** palette to build the front panel. 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.

3D and Classic Controls and Indicators

Many front panel objects have a high-color, three-dimensional appearance. Set your monitor to display at least 16-bit color for optimal appearance of the objects.

The 3D front panel objects also have corresponding low-color, two-dimensional objects. Use the 2D controls and indicators located on the **Classic Controls** palette to create VIs for 256-color and 16-color monitor settings.

Select **File»VI Properties** and select **Editor Options** from the **Category** pull-down menu to change the style of control or indicator LabVIEW creates when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu. Refer to the *Configuring the Appearance and Behavior of VIs* section of Chapter 16, *Customizing VIs*, for more information about configuring the appearance and behavior of VIs. Select **Tools»Options** and select **Front Panel** from the top pull-down menu to change the style of control or indicator LabVIEW creates in new VIs when you right-click a terminal and select **Create»Control** or **Create» Indicator** from the shortcut menu.

Slides, Knobs, Dials, Digital Displays, and Time Stamps

Use the numeric controls and indicators located on the **Numeric** and **Classic Numeric** palettes to simulate slides, knobs, dials, and digital displays. The palette also includes color boxes and color ramps for setting color values and a time stamp for setting the time and date for the data. Use numeric controls and indicators to enter and display numeric data.

Slide Controls and Indicators

The slide controls and indicators include vertical and horizontal slides, a tank, and a thermometer. Change the value of a slide control or indicator by using the Operating tool to drag the slider to a new position, by clicking a point of the slide object, or by using the optional digital display. If you drag the slider to a new position and the VI is running during the change, the control passes intermediate values to the VI, depending on how often the VI reads the control.

Slide controls or indicators can display more than one value. Right-click the object and select **Add Slider** from the shortcut menu to add more sliders. The data type of a control with multiple sliders is a cluster that contains each of the numeric values. Refer to the *Clusters* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about clusters.

Rotary Controls and Indicators

The rotary controls and indicators include knobs, dials, gauges, and meters. The rotary objects operate similarly to the slide controls and indicators. Change the value of a rotary control or indicator by moving the needles, by clicking a point of the rotary object, or by using the optional digital display.

Rotary controls or indicators can display more than one value. Right-click the object and select **Add Needle** from the shortcut menu to add new needles. The data type of a control with multiple needles is a cluster that contains each of the numeric values. Refer to the *Clusters* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about clusters.

Digital Controls and Indicators

Digital controls and indicators are the simplest way to enter and display numeric data. You can resize these front panel objects horizontally to accommodate more digits. You can change the value of a digital control or indicator by using the following methods:

- Use the Operating or Labeling tool to click inside the digital display window and enter numbers from the keyboard.
- Use the Operating tool to click the increment or decrement arrow buttons of a digital control.
- Use the Operating or Labeling tool to place the cursor to the right of the digit you want to change and press the up or down arrow key on the keyboard.

Numeric Formatting

By default, LabVIEW displays and stores numbers like a calculator. A numeric control or indicator displays up to six digits before automatically switching to exponential notation. You can configure the number of digits LabVIEW displays before switching to exponential notation in the **Format and Precision** tab of the **Numeric Properties** dialog box.

The precision you select affects only the display of the value. The internal accuracy still depends on the representation.

Time Stamp Control and Indicator

Use the time stamp control and indicator to send and retrieve a time and date value to or from the block diagram. You can change the value of the time stamp control by using the following methods:

- Right-click the constant and select **Format & Precision** from the shortcut menu.

- Click the **Time/Date Browse** button, shown at left, to display the **Set Time and Date** dialog box.
- Select **Data Operations»Set Time and Date** from the shortcut menu to display the **Set Time and Date** dialog box. You also can right-click the time stamp control and select **Data Operations»Set Time to Now** from the shortcut menu.

Color Boxes

A color box displays a color that corresponds to a specified value. For example, you can use color boxes to indicate different conditions, such as

out-of-range values. The color value is expressed as a hexadecimal number with the form RRGGBB. The first two digits control the red color value. The second two digits control the green color value. The last two digits control the blue color value.

Set the color of the color box by clicking it with the Operating or Coloring tool to display the color picker.

Color Ramps

A color ramp uses color to display its numeric value. You configure a color ramp that consists of at least two arbitrary markers, each with a numeric value and a corresponding display color. As the input value changes, the color displayed changes to the color that corresponds to that value. Color ramps are useful for visually indicating data ranges, such as a warning range for when a gauge reaches dangerous values. For example, you can use a color ramp to set the color scale for intensity charts and graphs. Refer to the *Intensity Graphs and Charts* section of Chapter 12, *Graphs and Charts*, for more information about intensity charts and graphs.

Right-click the color ramp and use the shortcut menu items to customize the appearance, size, colors, and number of colors.

You also can add a color ramp to any knob, dial, or gauge on the front panel. Meters have a visible color ramp by default.

Graphs and Charts

Use graphs and charts to display plots of data in a graphical form.

Refer to Chapter 12, *Graphs and Charts*, for more information about using graphs and charts in LabVIEW.

Buttons, Switches, and Lights

Use the Boolean controls and indicators to simulate buttons, switches, and lights. Use Boolean controls and indicators to enter and display Boolean (TRUE/FALSE) values. For example, if you are monitoring the temperature of an experiment, you can place a Boolean warning light on the front panel to indicate when the temperature goes above a certain level.

Use the shortcut menu to customize the appearance of Boolean objects and how they behave when you click them.

Text Entry Boxes, Labels, and Path Displays

Use the string and path controls and indicators to simulate text entry boxes and labels and to enter or return the location of a file or directory.

String Controls and Indicators

Enter or edit text in a string control on the front panel by using the Operating tool or the Labeling tool. By default, new or changed text does not pass to the block diagram until you terminate the edit session. You terminate the edit session by clicking elsewhere on the panel, changing to a different window, clicking the **Enter** button on the toolbar, or pressing the <Enter> key on the numeric keypad. Pressing the <Enter> key on the keyboard enters a carriage return.

Refer to the *Strings on the Front Panel* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about the string control and indicator.

Combo Box Controls

Use the combo box control to create a list of strings you can cycle through on the front panel. A combo box control is similar to a text or menu ring control. However, the value and data type of a combo box control are strings instead of numbers as with ring controls. Refer to the *Ring and Enumerated Type Controls and Indicators* section of this chapter for more information about ring controls.

You can use a combo box control to select the cases of a Case structure. Refer to the *Case Structures* section of Chapter 8, *Loops and Structures*, for more information about Case structures.

Right-click a combo box control and select **Edit Items** from the shortcut menu to add strings to the list from which you can select in the control. The order of the strings in the **Edit Items** page of the **Combo Box Properties** dialog box determines the order of the strings in the control. By default, the combo box control allows users to enter string values not already in the list of strings defined for the control. Right-click the combo box control and select **Allow Undefined Strings** from the shortcut menu to remove the checkmark and prevent the user from entering undefined string values in the control.

As you type a string in a combo box control at run time, LabVIEW selects the first, shortest string in the control that begins with the letters you type. If no strings match the letters you type and the control does not allow

undefined string values, LabVIEW does not accept or display the letters you type in the control.

When you configure the list of strings for a combo box control, you can specify a custom value for each string, which is useful if you want the string that appears in the combo box control on the front panel to be different than the string value the combo box terminal on the block diagram returns. Right-click the combo box control, select **Edit Items** from the shortcut menu, and remove the checkmark from the **Values Match Labels** checkbox in the **Edit Items** page of the **Combo Box Properties** dialog box. In the **Values** column of the table in this dialog box, change the value that corresponds to each string in the control.

Path Controls and Indicators

Use path controls and indicators to enter or return the location of a file or directory. Path controls and indicators work similarly to string controls and indicators, but LabVIEW formats the path using the standard syntax for the platform you are using.

Invalid Paths

If a function that returns a path fails, the function returns an invalid path value, `<Not A Path>`, in the indicator. Use the `<Not A Path>` value as the default value for a path control to detect when the user fails to provide a path and display a file dialog box with options for selecting a path. Use the File Dialog function to display a file dialog box.

Empty Paths

An empty path in a path control appears as an empty string on Windows and Mac OS and as a slash (`/`) on UNIX. Use empty paths to prompt the user to specify a path. When you wire an empty path to a file input/output (I/O) function, the empty path refers to the list of drives mapped to the computer.

(Mac OS) The empty path refers to the mounted volumes. **(UNIX)** The empty path refers to the root directory.

Array and Cluster Controls and Indicators

Use the array and cluster controls and indicators located on the **Array & Cluster** and **Classic Array & Cluster** palettes to create arrays and clusters of other controls and indicators. Refer to the *Grouping Data with Arrays*

and Clusters section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about arrays and clusters.

The **Array & Cluster** palettes also contain standard error cluster controls and indicators and the variant control. Refer to the *Error Clusters* section of Chapter 6, *Running and Debugging VIs*, for more information about error clusters. Refer to the *Handling Variant Data* section of Chapter 5, *Building the Block Diagram*, for more information about the variant control.

Listboxes, Tree Controls, and Tables

Use the listbox controls located on the **List & Table** and **Classic List & Table** palettes to give users a list of items from which to select.

Listboxes

You can configure listboxes to accept single or multiple selections. Use the multicolumn listbox to display more information about each item, such as the size of the item and the date it was created.

When you type characters in a listbox at run time, LabVIEW selects the first item in the listbox that begins with the characters you type. Use the left and right arrow keys to go to the previous or next items that match the characters you type.

You can add a symbol next to a list item, such as in the **VI Library Manager** dialog box, where directories and files have different symbols. You also can insert separator lines between list items.

You can use a Property Node to modify list items and to gather information about list items, such as to detect the currently selected items or to detect which items, if any, the user double-clicked. Refer to the *Property Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about Property Nodes.

Tree Controls

Use the tree control to give users a hierarchical list of items from which to select. You organize the items you enter in the tree control into groups of items, or nodes. Click the expand symbol next to a node to expand it and display all the items in that node. You also click the symbol next to the node to collapse the node.

Note You can create and edit tree controls only in the LabVIEW Full and Professional Development Systems. If a VI contains a tree control, you can run the VI in all LabVIEW packages, but you cannot configure the control in the Base Package.

When you type characters in a tree control at run time, LabVIEW selects the first item in the tree control that begins with the characters you type. You can change the hierarchy of items in the tree control by selecting an item and pressing the period (.) key to indent the current item or the comma (,) key to move the current item to the left.

You configure items in a tree control the same way you configure them in a listbox. You also can change the type of symbol that appears next to each node, and you can configure whether the user can drag items within the tree control.

You can use an Invoke Node to modify items in the tree control and to gather information about items, such as to detect which items, if any, the user double-clicked. When you add an item in the tree control, LabVIEW creates a unique tag for the item. You use this tag to modify items or to gather information about items programmatically. Right-click the tree control and select **Edit Items** from the shortcut menu to modify the tags that LabVIEW creates for each item. Refer to the *Invoke Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about Invoke Nodes.

Refer to the Directory Hierarchy in Tree Control VI in the `examples\general\controls\Directory Tree Control.llb` for an example of using a tree control.

Tables

Use the table control located on the **List & Table** and **Classic List & Table** palettes to create a table on the front panel.

Refer to the *Tables* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about using table controls.

Ring and Enumerated Type Controls and Indicators

Use the ring and enumerated type controls and indicators located on the **Ring & Enum** and **Classic Ring & Enum** palettes to create a list of strings you can cycle through.

Ring Controls

Ring controls are numeric objects that associate numeric values with strings or pictures. Ring controls appear as pull-down menus that users can cycle through to make selections.

Ring controls are useful for selecting mutually exclusive items, such as trigger modes. For example, use a ring control for users to select from continuous, single, and external triggering.

Right-click a ring control and select **Edit Items** from the shortcut menu to add items to the list from which you can select in the control. The order of the items in the **Edit Items** page of the **Ring Properties** dialog box determines the order of the items in the control. You also can configure the ring control so users can enter numeric values not already associated with any entries in the list of items defined for the control by right-clicking the ring control and select **Allow Undefined Values** from the shortcut menu.

To enter an undefined value in the ring control at run time, click the control, select **<Other>** from the shortcut menu, enter a numeric value in the digital display that appears, and press the <Enter> key. The undefined value appears in the ring control in angle brackets. LabVIEW does not add the undefined value to the list of items from which you can select in the control.

When you configure the list of items for a ring control, you can assign a specific numeric value to each item. If you do not assign specific numeric values to the items, LabVIEW assigns sequential values that correspond to the order of the items in the list, starting with a value of 0 for the first item. To assign specific numeric values, right-click the ring control, select **Edit Items** from the shortcut menu, and remove the checkmark from the **Sequential Values** checkbox in the **Edit Items** page of the **Ring Properties** dialog box. In the **Values** section of the table in this dialog box, change the numeric values that correspond to each item in the control. Each item in the ring control must have a unique numeric value.

Enumerated Type Controls

Use enumerated type controls to give users a list of items from which to select. An enumerated type control, or enum, is similar to a text or menu ring control. However, the data type of an enumerated type control includes information about the numeric values and the string labels in the control. The data type of a ring control is numeric.

Note You cannot allow the user to enter undefined values in enumerated type controls, and you cannot assign specific numeric values to items in enumerated type controls. If you need

this functionality, use a ring control. Refer to the *Ring Controls* section of this chapter for more information about ring controls.

You can use an enumerated type control to select the cases of a Case structure. Refer to the *Case Structures* section of Chapter 8, *Loops and Structures*, for more information about Case structures.

The numeric representation of the enumerated type control is an 8-bit, 16-bit, or 32-bit unsigned integer. Right-click the enumerated type control and select **Representation** from the shortcut menu to change the representation of the control.

Advanced Enumerated Type Controls and Indicators

All arithmetic functions except Increment and Decrement treat the enumerated type control the same as an unsigned integer. Increment increments the last enumerated value to the first, and Decrement decrements the first enumerated value to the last. When coercing a signed integer to an enumerated type, negative numbers are changed to equal the first enumerated value, and out-of-range positive numbers are changed to equal the last enumerated value. Out-of-range unsigned integers are always changed to equal the last enumerated value.

If you wire a floating-point value to an enumerated type indicator, LabVIEW coerces the floating-point value to the closest numeric value in the enumerated type indicator. LabVIEW handles out-of-range numbers as previously described. If you wire an enumerated control to any numeric value, LabVIEW coerces the enumerated type value to a numeric value. To wire an enumerated type control to an enumerated type indicator, the items in the indicator must match the items in the control. However, the indicator can have more items than the control.

Container Controls

Use the container controls located on the **Containers** and the **Classic Containers** palettes to group controls and indicators, to display the front panel of another VI on the front panel of the current VI, or **(Windows)** to display ActiveX objects on the front panel. Refer to Chapter 19, *Windows Connectivity*, of this manual for more information about using ActiveX.

Tab Controls

Use tab controls to overlap front panel controls and indicators in a smaller area. A tab control consists of pages and tabs. Place front panel objects on

each page of a tab control and use the tab as the selector for displaying different pages.

Tab controls are useful when you have several front panel objects that are used together or during a specific phase of operation. For example, you might have a VI that requires the user to first configure several settings before a test can start, then allows the user to modify aspects of the test as it progresses, and finally allows the user to display and store only pertinent data.

On the block diagram, the tab control is an enumerated type control by default. Terminals for controls and indicators placed on the tab control appear as any other block diagram terminal. Refer to the *Enumerated Type Controls* section of this chapter for more information about enumerated type controls.

Subpanel Controls

Use the subpanel control to display the front panel of another VI on the front panel of the current VI. For example, you can use a subpanel control to design a user interface that behaves like a wizard. Place the **Back** and **Next** buttons on the front panel of the top-level VI and use a subpanel control to load different front panels for each step of the wizard.

Note You can create and edit subpanel controls only in the LabVIEW Full and Professional Development Systems. If a VI contains a subpanel control, you can run the VI in all LabVIEW packages, but you cannot configure the control in the Base Package.

When you place a subpanel control on the front panel, LabVIEW does not create a front panel terminal for the control on the block diagram. Instead, LabVIEW creates an Invoke Node on the block diagram with the Insert VI method selected. To load a VI in the subpanel control, wire a reference to that VI to the Invoke Node. Refer to Chapter 17, *Programmatically Controlling VIs*, of this manual for more information about using VI references and Invoke Nodes.

Note Because the subpanel control does not have a terminal, you cannot create an array of subpanel controls, and you cannot create a type definition of a subpanel control. You can place a subpanel control in a cluster to group the subpanel control with other controls, but the cluster cannot contain only a subpanel control or controls.

If the front panel of the VI you want to load is open or if you loaded the front panel in another subpanel control on the same front panel, LabVIEW returns an error, and you cannot load the front panel in the subpanel control.

You also cannot load the front panel of a VI in a remote instance of LabVIEW, and you cannot load front panels recursively.

You cannot use keyboard shortcuts to navigate or operate the front panel in the subpanel control.

If you load a VI that is not running, the VI in the subpanel control loads in run mode.

LabVIEW displays only the visible area of the front panel of the VI you load in the subpanel control. After you stop the VI that contains the subpanel control, LabVIEW clears the front panel in the subpanel control. You also can use the Remove VI method to unload the VI in the subpanel control.

Refer to the `examples\general\controls\subpanel.llb` for examples of using subpanel controls.

I/O Name Controls and Indicators

Use the I/O name controls and indicators to pass DAQ channel names, VISA resource names, and IVI logical names you configure to I/O VIs to communicate with an instrument or a DAQ device.

I/O name constants are located on the **Functions** palette.

Note All I/O name controls or constants are available on all platforms. This allows you to develop I/O VIs on any platform that can communicate with devices that are platform specific. However, if you try to run a VI with a platform-specific I/O control on a platform that does not support that device, you will receive an error.

(Windows) Use Measurement & Automation Explorer, available from the **Tools** menu, to configure DAQ channel names, VISA resource names, and IVI logical names.

(Mac OS) Use the NI-DAQ Configuration Utility, available from the **Tools** menu, to configure National Instruments DAQ hardware. Use the DAQ Channel Wizard, available from the **Tools** menu, to configure DAQ channel names.

(Mac OS and UNIX) Use the configuration utilities for your instrument to configure VISA resource names and IVI logical names. Refer to the documentation for your instrument for more information about the configuration utilities.

The IMAQ session control is a unique identifier that represents the connection to the hardware.

Waveform Control

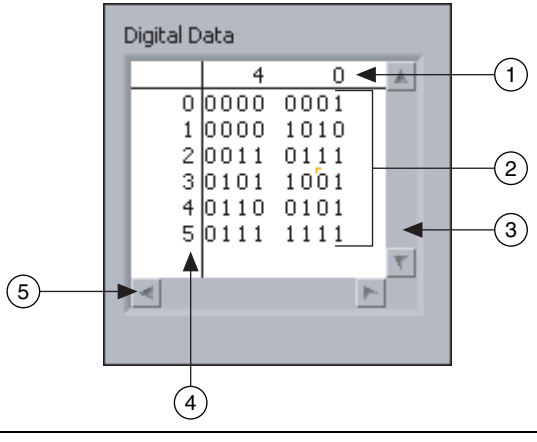
Use the waveform control to manipulate individual data elements of a waveform. Refer to the *Waveform Data Type* section of Chapter 12, *Graphs and Charts*, for more information about the waveform data type.

Digital Waveform Control

Use the Digital Waveform control to manipulate the individual elements of a digital waveform. Refer to the *Digital Waveform Data Type* section of Chapter 12, *Graphs and Charts*, for more information about the Digital Waveform control.

Digital Data Control

The digital data control includes digital data arranged in rows and columns. Use the digital data type to build digital waveforms or to display digital data extracted from a digital waveform.Wire the digital waveform data type to a digital data indicator to view the samples and signals of a digital waveform. The digital data control in Figure 4-2 displays five samples that each contain eight signals.



1	Signals	3	Vertical Scroll Bar	5	Horizontal Scroll Bar
2	Data	4	Samples		

Figure 4-2. Digital Data Control

You can insert and delete rows and columns in the digital data control. To insert a row, right-click a sample in the sample column and select **Insert Row Before** from the shortcut menu. To delete a row, right-click a sample in the sample column and select **Delete Row** from the shortcut menu. To insert a column, right-click a signal in the signal column and select **Insert Column Before** from the shortcut menu. To delete a column, right-click a signal in the signal column and select **Delete Column** from the shortcut menu.

You also can cut, copy, and paste digital data within the control. To cut data, select the row or column you want to cut, right-click, and select **Data Operations»Cut Data** from the shortcut menu. You can cut only whole rows or columns of data. You cannot create a new row or column with the digital data you cut. To copy data, select the area you want to copy, right-click, and select **Data Operations»Copy Data** from the shortcut menu. To paste digital data, select the area you want to copy into and select **Data Operations»Paste Data** from the shortcut menu. You must paste digital data into an area that is the same dimension as the area you cut or copied from. For example, if you copy four bits of data from one row, you must select four existing bits of data to paste over in the same row or in a different row. If you copy four bits of data from an area of two rows by two columns, you must paste the data into an area of two rows by two columns.

The digital data control and the digital waveform control accept values of 0, 1, L, H, Z, X, T, and V. You can display the data in the digital data control in binary, hexadecimal, octal, and decimal formats. The digital states L, H, Z, X, T, and V, which are states some measurement devices use, appear as question marks if you choose to display the values in hexadecimal, octal, or decimal format. Right-click the control, select **Data Format** from the shortcut menu, and select a data format for the control.

Converting Data to Digital Data

In most cases, a signal you acquire is returned as raw data. To graph the signal in a digital waveform graph, you must convert the raw data you acquire into the digital data type or the digital waveform data type. Use the Analog to Digital Waveform VI to convert the data to the digital waveform data type. Use the Get Waveform Components function to extract the digital data from the digital waveform data type.

The block diagram in Figure 4-3 simulates the acquisition of a sine wave with an amplitude of 5, which means the sine wave can contain values that range from `-5` to `5`. The Analog to Digital Waveform VI in this block diagram represents each value with 8 bits. The 8 bits can represent a

minimum value of –5 and a maximum value of 5. The Digital Waveform probe displays a portion of the resulting values in binary format.

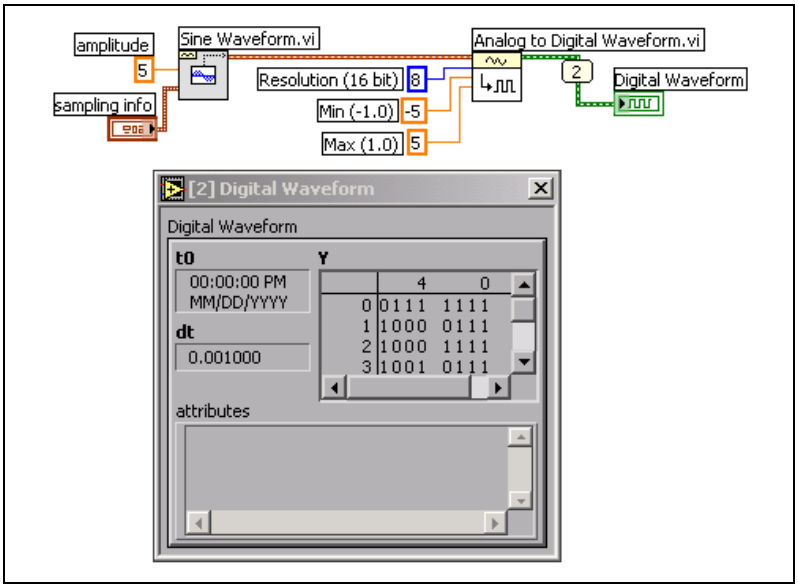


Figure 4-3. Converting an Analog Waveform to a Digital Waveform

Acquiring a Digital Subset

Use the Digital Signal Subset VI to extract individual signals from a digital waveform. The block diagram in Figure 4-4 shows how you can combine extracted individual signals with other digital data to create new digital waveforms.

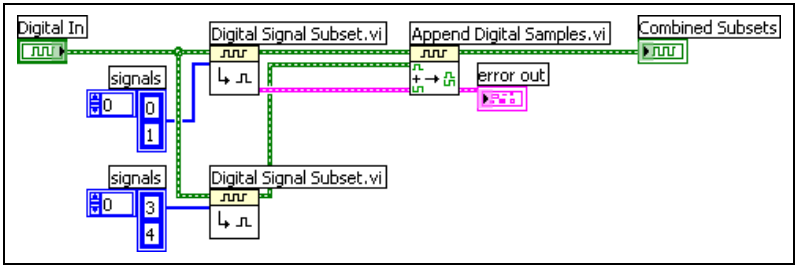


Figure 4-4. Appending Two Digital Waveforms

The top Digital Signal Subset VI extracts the first and second signals from the digital waveform. The bottom Digital Signal Subset VI extracts the fourth and fifth signals. The Append Digital Samples VI appends the first signal to the fourth signal, appends the second signal to the fifth signal, and plots the resulting two signals on a digital waveform graph.

Appending Digital Samples and Signals

Use the Append Digital Signals VI to append the samples in one digital signal to the end of samples of another digital signal. You can append signals with the same number of samples or of different numbers of samples. For example, if you have two digital signals that both consist of two rows of 8 bits, the resulting digital data consists of two rows of 16 bits. If you combine two signals, one that consists of two rows of 8 bits and the other that consists of one row of 8 bits, the resulting digital data consists of two rows of 16 bits. The Append Digital Signals VI pads the remaining columns in the second sample with the value you select in the **default value** input.

Compressing Digital Data

Use the Compress Digital Data VI to compress digital data when you want to display two or more serial digital signals with the same bit sets on the same row to better visualize the data. For example, if you acquire 10 digital waveform samples and the tenth waveform differs from the other nine, compressing the digital data helps you easily find which waveform is different. Compressing digital data also conserves memory resources. Use the Uncompress Digital Data VI to uncompress digital data you compress.

Searching for a Pattern

Use the Search for Digital Pattern VI to specify a digital pattern you want to search for. For example, if you acquire a large digital waveform and want to see if any part of the digital waveform matches a certain pattern, wire that pattern to the **digital pattern** input of the Search for Digital Pattern VI to discover any matches.

References to Objects or Applications

Use the reference number controls and indicators located on the **Refnum** and **Classic Refnum** palettes to work with files, directories, devices, and network connections. Use the control refnum to pass front panel object information to subVIs. Refer to the *Controlling Front Panel Objects* section of Chapter 17, *Programmatically Controlling VIs*, for more information about control refnums.

A reference number, or refnum, is a unique identifier for an object, such as a file, device, or network connection. 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 or indicator to pass a refnum into or out of a VI. For example, use a refnum control or indicator 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 object, it is valid only for the period during which the object is open. If you close the object, LabVIEW disassociates the refnum with the object, and the refnum becomes obsolete. If you open the object again, LabVIEW creates a new refnum that is different from the first refnum.

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 object. If a VI opens an object multiple times, each open operation returns a different refnum.

Dialog Controls and Indicators

Use the dialog controls in dialog boxes you create. The dialog controls and indicators are designed specifically for use in dialog boxes and include ring and spin controls, numeric slides and progress bars, listboxes, tables, string and path controls, tab controls, tree controls, buttons, checkboxes, radio buttons, and an opaque label that automatically matches the background color of its parent. These controls differ from those that appear on the front panel only in terms of appearance. These controls appear in the colors you have set up for your system.

Because the dialog controls change appearance depending on which platform you run the VI, the appearance of controls in VIs you create is compatible on all LabVIEW platforms. When you run the VI on a different

platform, the dialog controls adapt their color and appearance to match the standard dialog box controls for that platform.

Select **File»VI Properties** and select **Window Appearance** from the **Category** pull-down menu to hide the menu bar and scrollbars and to create VIs that look and behave like standard dialog boxes for each platform. Select **Editor Options** from the **Category** pull-down menu to change the style of control or indicator LabVIEW creates when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu. Select **Tools»Options** and select **Front Panel** from the top pull-down menu to change the style of control or indicator LabVIEW creates in new VIs when you right-click a terminal and select **Create» Control** or **Create»Indicator** from the shortcut menu. Refer to the *Configuring the Appearance and Behavior of VIs* section of Chapter 16, *Customizing VIs*, for more information about configuring the appearance and behavior of VIs.

Labeling

Use labels to identify objects on the front panel and block diagram.

LabVIEW includes two kinds of labels—owned labels and free labels. Owned labels belong to and move with a particular object and annotate that object only. You can move an owned label independently, but when you move the object that owns the label, the label moves with the object. You can hide owned labels, but you cannot copy or delete them independently of their owners. You also can display a unit label for numeric controls and indicators by selecting **Visible Items»Unit Label** from the shortcut menu. Refer to the *Numeric Units and Strict Type Checking* section of Chapter 5, *Building the Block Diagram*, for more information about numeric units.

Free labels are not attached to any object, and you can create, move, rotate, or delete them independently. Use them to annotate front panels and block diagrams.

Free labels are useful for documenting code on the block diagram and for listing user instructions on the front panel. Double-click an open space or use the Labeling tool to create free labels or to edit either type of label.

Refer to the *Labels* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about creating descriptive labels for a user interface.

Captions

Front panel objects also can have captions. Right-click the object and select **Visible Items»Caption** from the shortcut menu to display the caption. Unlike a label, a caption does not affect the name of the object, and you can use it as a more descriptive object label. The caption does not appear on the block diagram.

If you assign the object to a connector pane terminal, the caption appears when you use the Wiring tool to move the cursor over the terminal on the block diagram. The caption also appears next to the terminal in the **Context Help** window. Refer to the *Setting up the Connector Pane* section of Chapter 7, *Creating VIs and SubVIs*, for more information about connector pane terminals.

Text Characteristics

LabVIEW uses fonts already installed on your computer. Use the **Text Settings** pull-down menu on the toolbar to change the attributes of text. If you select objects or text before you make a selection from the **Text Settings** pull-down menu, the changes apply to everything you select. If you select nothing, the changes apply to the default font. Changing the default font does not change the font of existing labels. It affects only those labels you create from that point on.

Select **Font Dialog** from the **Text Settings** pull-down menu on the front panel to apply specific font styles to text you have selected. If you do not select any text, the **Panel Default** option contains a checkmark. If you select **Text Settings»Font Dialog** from the block diagram without selecting any objects, the **Diagram Default** option contains a checkmark. You can set different fonts for the front panel and for the block diagram. For example, you can have a small font on the block diagram and a large one on the front panel.

The **Text Settings** pull-down menu contains the following built-in fonts:

- **Application Font**—Default font used for **Controls** and **Functions** palettes and text in new controls
- **System Font**—Used for menus
- **Dialog Font**—Used for text in dialog boxes

When you transfer a VI that contains one of these built-in fonts to another platform, the fonts correspond as closely as possible.

The **Text Settings** pull-down menu also has **Size**, **Style**, **Justify**, and **Color** submenu items.

Font selections you make from any of these submenus apply to objects you selected. For example, if you select a new font while you have a knob and a graph selected, the labels, scales, and digital displays all change to the new font.

LabVIEW preserves as many font attributes as possible when you make a change. For example, if you change several objects to the Courier font, the objects retain their size and styles if possible. When you use the **Font** dialog box, LabVIEW changes the objects you select to the text characteristics you select. If you select one of the built-in fonts or the current font, LabVIEW changes the selected objects to the font and size associated with that font.

When you work with objects that have multiple pieces of text, like slides, font changes you make affect the objects or text you currently have selected. For example, if you select the entire slide and select **Style»Bold** from the **Text Settings** pull-down menu, the scale, digital display, and label all change to a bold font. If you select only the label and select **Bold**, only the label changes to a bold font. If you select text from a scale marker and select **Bold**, all the markers change to a bold font.

Refer to the *Fonts and Text Characteristics* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about using fonts and text characteristics to design a user interface.

Designing User Interfaces

If a VI serves as a user interface or a dialog box, front panel appearance and layout are important. Design the front panel so users can easily identify what actions to perform. You can design front panels that look similar to instruments or other devices.

Refer to the *LabVIEW Development Guidelines* manual for more information about designing user interfaces.

Refer to the *Case and Sequence Structures* section of Chapter 8, *Loops and Structures*, for information about using events to enhance the functionality of user interfaces.

Using Front Panel Controls and Indicators

Controls and indicators are the main components of the front panel. When you design the front panel, consider how users interact with the VI and group controls and indicators logically. If several controls are related, add a decorative border around them or put them in a cluster. Use the decorations located on the **Decorations** palette to group or separate objects on a front panel with boxes, lines, or arrows. These objects are for decoration only and do not display data.

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

Assign specific names to buttons and use common terminology. Use names like Start, Stop, and Save instead of OK. Specific names make it easier for users to use the VI.

Use the default LabVIEW fonts and colors. LabVIEW replaces the built-in fonts with comparable font families on different platforms. If you select a different font, LabVIEW substitutes the closest match if the font is unavailable on a computer. LabVIEW handles colors similarly to fonts. If a color is not available on a computer, LabVIEW replaces it with the closest match. You also can use system colors to adapt the appearance of a front panel to the system colors of any computer that runs the VI.

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

Refer to Chapter 6, *LabVIEW Style Guide*, of the *LabVIEW Development Guidelines* manual for more information about using layout, fonts, and color to design a user interface.

Designing Dialog Boxes

If a VI contains consecutive dialog boxes that appear in the same screen location, organize them so that the buttons in the first dialog box do not directly line up with the buttons in the next dialog box. Users might double-click a button in the first dialog box and unknowingly click a button in the subsequent dialog box.

Refer to the *Dialog Controls and Indicators* section of this chapter for more information about the dialog controls. Refer to the *Dialog Boxes* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development*

Guidelines manual for more information about using dialog boxes in a user interface.

Selecting the Screen Size

When you design a VI, consider whether the front panel can display on computers with different screen resolutions. Select **File»VI Properties**, select **Window Size** in the **Category** pull-down menu, and place a checkmark in the **Maintain Proportions of Window for Different Monitor Resolutions** checkbox to maintain front panel window proportions relative to the screen resolution.

Refer to the *Sizing and Positioning* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about selecting the screen size of a user interface.

5

Building the Block Diagram

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

For more information…

Refer to the *LabVIEW Help* for more information about designing and configuring the block diagram.

Relationship between Front Panel Objects and Block Diagram Terminals

Front panel objects appear as terminals on the block diagram. Double-click a block diagram terminal to highlight the corresponding control or indicator on the front panel.

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

Block Diagram Objects

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

Block Diagram Terminals

You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a knob icon terminal, shown at left, represents a knob on the front panel. The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric.

A DBL terminal, shown at left, represents a double-precision, floating-point numeric control or indicator. Right-click a terminal and select **Display Icon** from the shortcut menu to remove the checkmark and to display the data type for the terminal. Use icon terminals to display the types of front panel objects on the block diagram, in addition to the data types of the front panel objects. Use data type terminals to conserve space on the block diagram.

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

A terminal is any point to which you can attach a wire, other than to another wire. LabVIEW has control and indicator terminals, node terminals, constants, and specialized terminals on structures, such as the input and output terminals on the Formula Node. You use wires to connect terminals and pass data to other terminals. Right-click a block diagram object and select **Visible Items»Terminals** from the shortcut menu to view the terminals. Right-click the object and select **Visible Items»Terminals** again to hide the terminals. This shortcut menu item is not available for expandable VIs and functions.

Control and Indicator Data Types

Table 5-1 shows the symbols for the different types of control and indicator terminals. The color and symbol of each terminal indicate the data type of the control or indicator. Control terminals have a thicker border than indicator terminals. Also, arrows appear on front panel terminals to indicate whether the terminal is a control or an indicator. An arrow appears on the right if the terminal is a control, and an arrow appears on the left if the terminal is an indicator.

Table 5-1. Control and Indicator Terminals

Control	Indicator	Data Type	Color	Default Values
SGL	SGL	Single-precision floating-point numeric	Orange	0.0
DBL	DBL	Double-precision floating-point numeric	Orange	0.0
EXT	EXT	Extended-precision floating-point numeric	Orange	0.0
CSG	CSG	Complex single-precision floating-point numeric	Orange	0.0 + i0.0

Table 5-1. Control and Indicator Terminals (Continued)

Control	Indicator	Data Type	Color	Default Values
		Complex double-precision floating-point numeric	Orange	0.0 + i0.0
		Complex extended-precision floating-point numeric	Orange	0.0 + i0.0
		8-bit signed integer numeric	Blue	0
		16-bit signed integer numeric	Blue	0
		32-bit signed integer numeric	Blue	0
		8-bit unsigned integer numeric	Blue	0
		16-bit unsigned integer numeric	Blue	0
		32-bit unsigned integer numeric	Blue	0
		<64.64>-bit time stamp	Brown	date and time (local)
		Enumerated type	Blue	—
		Boolean	Green	FALSE
		String	Pink	empty string
		Array—Encloses the data type of its elements in square brackets and takes the color of that data type.	Varies	—
 	 	Cluster—Encloses several data types. Cluster data types are brown if all elements of the cluster are numeric or pink if the elements of the cluster are different types.	Brown or Pink	—
		Path	Aqua	<Not A Path>
		Dynamic	Blue	—

Table 5-1. Control and Indicator Terminals (Continued)

Control	Indicator	Data Type	Color	Default Values
		Waveform—Cluster of elements that carries the data, start time, and Δt of a waveform. Refer to the *Waveform Data Type* section of Chapter 12, *Graphs and Charts*, for more information about the waveform data type.	Brown	—
		Digital waveform	Dark Green	—
		Digital data	Dark Green	—
		Reference number (refnum)	Aqua	—
		Variant—Stores the control or indicator name, information about the data type from which you converted, and the data itself. Refer to the *Handling Variant Data* section of this chapter for more information about the variant data type.	Purple	—
		I/O name—Passes DAQ channel names, VISA resource names, and IVI logical names you configure to I/O VIs to communicate with an instrument or a DAQ device. Refer to the *I/O Name Controls and Indicators* section of Chapter 4, *Building the Front Panel*, for more information about the I/O name data type.	Purple	—
		Picture—Displays pictures that can contain lines, circles, text, and other types of graphic shapes. Refer to the *Using the Picture Indicator* section of Chapter 13, *Graphics and Sound VIs*, for more information about the picture data type.	Blue	—

Many data types have a corresponding set of functions that can manipulate the data. Refer to the *Functions Overview* section of this chapter for information about which functions to use with each data type.

Refer to the *Memory and Speed Optimization* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about selecting a data type to optimize memory usage.

Constants

Constants are terminals on the block diagram that supply fixed data values to the block diagram. Universal constants are constants with fixed values, such as pi (π) and infinity (∞). User-defined constants are constants you define and edit before you run a VI.

Label a constant by right-clicking the constant and selecting **Visible Items»Label** from the shortcut menu. Universal constants have predetermined values for their labels that you can edit by using the Operating tool or the Labeling tool.

Most constants are located at the bottom or top of their palettes.

Universal Constants

Use universal constants for mathematical computations and formatting strings or paths. LabVIEW includes the following types of universal constants:

- **Universal numeric constants**—A set of high-precision and commonly used mathematical and physical values, such as the natural logarithm base (e) and the speed of light. The universal numeric constants are located on the **Additional Numeric Constants** palette.
- **Universal string constants**—A set of commonly used non-displayable string characters, such as line feed and carriage return. The universal string constants are located on the **String** palette.
- **Universal file constants**—A set of commonly used file path values, such as Not a Path, Not a Refnum, and Default Directory. The universal file constants are located on the **File Constants** palette.

User-Defined Constants

The **Functions** palette includes constants organized by type, such as Boolean, numeric, ring, enumerated type, color box, string, array, cluster, and path constants.

Create a user-defined constant by right-clicking an input terminal of a VI or function and selecting **Create Constant** from the shortcut menu. You cannot change the value of user-defined constants when the VI is running.

You also can create a constant by dragging a front panel control to the block diagram. LabVIEW creates a constant that contains the value of the front panel control at the time you dragged it to the block diagram. The front panel control remains on the front panel. Changing the value of the control does not affect the constant value and vice versa.

Use the Operating or Labeling tool to click the constant and edit its value. If automatic tool selection is enabled, double-click the constant to switch to the Labeling tool and edit the value.

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. LabVIEW includes the following types of nodes:

- **Functions**—Built-in execution elements, comparable to an operator, function, or statement. Refer to the *Functions Overview* section of this chapter for more information about the functions available in LabVIEW.
- **SubVIs**—VIs used on the block diagram of another VI, comparable to subroutines. Refer to the *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about using subVIs on the block diagram.
- **Structures**—Process control elements, such as Flat and Stacked Sequence structures, Case structures, For Loops, or While Loops. Refer to Chapter 8, *Loops and Structures*, for more information about using structures.
- **Formula Nodes**—Resizable structures for entering equations directly into a block diagram. Refer to the *Formula Nodes* section of Chapter 21, *Formulas and Equations*, for more information about using Formula Nodes.
- **Expression Nodes**—Structures for calculating expressions, or equations, that contain a single variable. Refer to the *Expression Nodes* section of Chapter 21, *Formulas and Equations*, for more information about using Expression Nodes.
- **Property Nodes**—Structures for setting or finding properties of a class. Refer to the *Property Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using Property Nodes.
- **Invoke Nodes**—Structures for executing methods of a class. Refer to the *Invoke Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using Invoke Nodes.
- **Code Interface Nodes (CINs)**—Structures for calling code from text-based programming languages. Refer to the *Code Interface Node* section of Chapter 20, *Calling Code from Text-Based Programming*

Languages, for more information about calling code from text-based programming languages.

- **Call By Reference Nodes**—Structures for calling a dynamically loaded VI. Refer to the *Call By Reference Nodes and Strictly Typed VI Refnums* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using Call By Reference Nodes.
- **Call Library Nodes**—Structures for calling most standard shared libraries or DLLs. Refer to the *Call Library Function Node* section of Chapter 20, *Calling Code from Text-Based Programming Languages*, for more information about using Call Library Nodes.

Functions Overview

Functions are the essential operating elements of LabVIEW. Function icons on the **Functions** palette have pale yellow backgrounds and black foregrounds. Functions do not have front panels or block diagrams but do have connector panes. You cannot open nor edit a function.

The **Functions** palette also includes the VIs that ship with LabVIEW. Use these VIs as subVIs when you build data acquisition, instrument control, communication, and other VIs. Refer to the *Using Built-In VIs and Functions* section of Chapter 7, *Creating VIs and SubVIs*, for more information about using the built-in VIs.

Numeric Functions

Use the Numeric functions to create and perform arithmetic, trigonometric, logarithmic, and complex mathematical operations on numbers and to convert numbers from one data type to another.

Boolean Functions

Use the Boolean functions to perform logical operations on single Boolean values or arrays of Boolean values, such as the following tasks:

- Change a TRUE value to a FALSE value and vice versa.
- Determine which Boolean value to return if you receive two or more Boolean values.
- Convert a Boolean value to a number (either 1 or 0).
- Perform compound arithmetic on two or more Boolean values.

String Functions

Use the String functions to perform the following tasks:

- Concatenate two or more strings.
- Extract a subset of strings from a string.
- Search for and replace characters or subsets of strings in a string.
- Convert numeric data into strings.
- Format a string for use in a word processing or spreadsheet application.

Refer to the *Strings* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about using the String functions.

Array Functions

Use the Array functions to create and manipulate arrays, such as the following tasks:

- Extract individual data elements from an array.
- Add individual data elements to an array.
- Split arrays.

Refer to the *Arrays* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about using the Array functions.

Cluster Functions

Use the Cluster functions to create and manipulate clusters, such as the following tasks:

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

Refer to the *Clusters* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about using the Cluster functions.

Comparison Functions

Use the Comparison functions to compare Boolean values, strings, numerics, arrays, and clusters.

Refer to Appendix C, *Comparison Functions*, for more information about using the Comparison functions.

Time and Dialog Functions

Use the Time and Dialog functions to perform the following tasks:

- Manipulate the speed at which an operation executes.
- Retrieve time and date information from your computer clock.
- Create dialog boxes to prompt users with instructions.

The **Time & Dialog** palette also includes the Error Handler VIs. Refer to the *Error Checking and Error Handling* section of Chapter 6, *Running and Debugging VIs*, for more information about using the Error Handler VIs.

File I/O Functions

Use the File I/O functions to perform the following tasks:

- Open and close files.
- Read from and write to files.
- Create directories and files you specify in the path control.
- Retrieve directory information.
- Write strings, numbers, arrays, and clusters to files.

The **File I/O** palette also includes VIs that perform common file I/O tasks. Refer to Chapter 14, *File I/O*, for more information about using the File I/O VIs and functions.

Waveform Functions

Use the Waveform functions to perform the following tasks:

- Build waveforms that include the waveform values, channel information, and timing information.
- Extract individual data elements from a waveform.
- Edit individual data elements of a waveform.

Refer to Chapter 5, *Creating a Typical Measurement Application*, of the *LabVIEW Measurements Manual* for more information about creating and using waveforms in VIs.

Application Control Functions

Use the Application Control functions to programmatically control VIs and LabVIEW applications on your local computer or across a network. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using the Application Control functions.

Advanced Functions

Use the Advanced functions to call code from libraries, such as dynamic link libraries (DLLs), to manipulate LabVIEW data for use in other applications, to create and manipulate Windows registry keys, and to call a section of code from text-based programming languages. Refer to the *Using External Code in LabVIEW* manual for more information about using the Advanced functions.

Adding Terminals to Functions

You can change the number of terminals for some functions. For example, to build an array with 10 elements, you must add 10 terminals to the Build Array function.

You can add terminals to expandable VIs and functions by using the Positioning tool to drag the top or bottom borders of the function up or down, respectively. You also can use the Positioning tool to remove terminals from expandable VIs and functions, but you cannot remove a terminal that is already wired. You must first delete the existing wire to remove the terminal.

You also can add or remove terminals by right-clicking one of the terminals of the function and selecting **Add Input**, **Add Output**, **Remove Input**, or **Remove Output** from the shortcut menu. Depending on the function, you can add terminals for inputs, outputs, or refnum controls. The **Add Input** and **Add Output** shortcut menu items add a terminal immediately after the terminal you right-clicked. The **Remove Input** and **Remove Output** shortcut menu items remove the terminal you right-clicked. If you use the shortcut menu items to remove a wired terminal, LabVIEW removes the terminal and disconnects the wire.

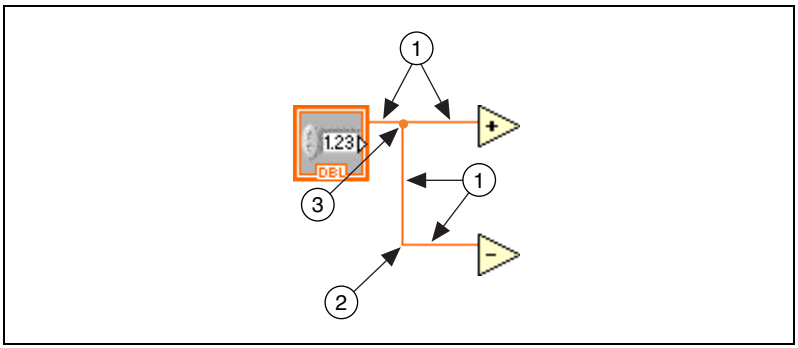
Using Wires to Link Block Diagram Objects

You transfer data among block diagram objects through wires. Each wire has a single data source, but you can wire it to many VIs and functions that read the data.You must wire all required block diagram terminals. Otherwise, the VI is broken and will not run. Display the **Context Help** window to see which terminals a block diagram node requires. The labels of required terminals appear bold in the **Context Help** window. Refer to the *Setting Required, Recommended, and Optional Inputs and Outputs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about required terminals. Refer to the *Correcting Broken VIs* section of Chapter 6, *Running and Debugging VIs*, for more information about broken VIs.

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. The arrows on either side of the red X indicate the direction of the data flow, and the color of the arrows indicate the data type of the data flowing through the wire. Refer to the *LabVIEW Quick Reference Card* for more information about wire data types.

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

A wire segment is a single horizontal or vertical piece of wire. A bend in a wire is where two segments join. The point at which two or more wire segments join is a junction. A wire branch contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions in between. Figure 5-1 shows a wire segment, bend, and junction.



1 Segment 2 Bend 3 Junction

Figure 5-1. Wire Segment, Bend, and Junction

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

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

(Mac OS) Press the <Option> key and click. **(UNIX and Linux)** Click the middle mouse button.

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

Caution Crossing wires can clutter a block diagram and make the block diagram difficult to debug.

Refer to the *Wiring Techniques* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about wiring tips and techniques.

Automatically Wiring Objects

LabVIEW automatically wires objects as you place them on the block diagram. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and leaves terminals that do not match unconnected.

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.

Toggle automatic wiring by pressing the space bar 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.

(Mac OS) Press the <Option> key. **(Sun)** Press the <Meta> key. **(Linux)** Press the <Alt> key.

You can disable automatic wiring by selecting **Tools»Options** and selecting **Block Diagram** from the top pull-down menu.

Manually Wiring Objects

Use the Wiring tool to manually connect the terminals on one block diagram node to the terminals on another block diagram node. The cursor point of the tool is the tip of the unwound wire spool. When you move the Wiring tool over a terminal, the terminal blinks. When you move the Wiring tool over a VI or function terminal, a tip strip also appears, listing the name of the terminal. If wiring to the terminal would create a broken wire, the cursor changes to a wire spool with a warning symbol. You can create the broken wire, but you must correct the broken wire before you can run the VI. Refer to the *Correcting Broken Wires* section of this chapter for more information about correcting broken wires.

Use the **Context Help** window to determine exactly where to connect wires. When you move the cursor over a VI or function, the **Context Help** window lists each terminal of the VI or function. The **Context Help** window does not display terminals for expandable VIs and functions, such as the Build Array function. Click the **Show Optional Terminals and Full Path** button in the **Context Help** window to display the optional terminals of the connector pane. Refer to the *Setting Required, Recommended, and Optional Inputs and Outputs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about optional terminals.

Routing Wires

LabVIEW automatically finds a route for a wire as you wire it. LabVIEW routes a wire around existing objects on the block diagram, such as loops and structures. LabVIEW also routes a wire to decrease the number of bends in the wire. When possible, automatically routed wires from control terminals exit the right side of the terminal, and automatically routed wires to indicator terminals enter the left side of the terminal.

To automatically route an existing wire, right-click the wire and select **Clean Up Wire** from the shortcut menu.

Press the <A> key after you start a wire to temporarily disable automatic wire routing and route a wire manually. Press the <A> key again to enable automatic wire routing for the wire. After you end the wire, LabVIEW enables automatic wire routing again. You also can temporarily disable automatic routing after you click to start or set a wire by holding down the mouse button while you wire to another terminal or set point and then releasing the mouse button. After you release the mouse button, LabVIEW enables automatic wire routing again.

You can disable automatic wire routing for all new wires by selecting **Tools»Options** and selecting **Block Diagram** from the top pull-down menu.

If you disable automatic wire routing, you can wire terminals vertically or horizontally depending on the direction in which you first move the Wiring tool. The wire connects to the center of the terminal, regardless of where you click the terminal. After you click the terminal, press the spacebar to switch between the horizontal and vertical direction.

You also can press the spacebar to switch between the horizontal and vertical direction if automatic wire routing is enabled. If LabVIEW finds a route for the wire in the new direction, the wire switches to that direction.

Selecting Wires

Select wires by using the Positioning tool to single-click, double-click, or triple-click them. Single-clicking a wire selects one segment of the wire. Double-clicking a wire selects a branch. Triple-clicking a wire selects the entire wire.

Correcting Broken Wires

A broken wire appears as a dashed black line with a red `X` in the middle. Broken wires occur for a variety of reasons, such as when you try to wire

two objects with incompatible data types. Move the Wiring tool over a broken wire to display a tip strip that describes why the wire is broken. This information also appears in the **Context Help** window when you move the Wiring tool over a broken wire. Right-click the wire and select **List Errors** from the shortcut menu to display the **Error list** window. Click the **Help** button for more information about why the wire is broken.

Triple-click the wire with the Positioning tool and press the <Delete> key to remove a broken wire. You also can right-click the wire and select from shortcut menu options such as **Delete Wire Branch**, **Create Wire Branch**, **Remove Loose Ends**, **Clean Up Wire**, **Change to Control**, **Change to Indicator**, **Enable Indexing at Source**, and **Disable Indexing at Source**. These options change depending on the reason for the broken wire.

You can remove all broken wires by selecting **Edit»Remove Broken Wires** or by pressing the <Ctrl-B> keys. **(Mac OS)** Press the <Command-B> keys. **(UNIX)** Press the <Meta-B> keys.

Caution Use caution when removing all broken wires. Sometimes a wire appears broken because you are not finished wiring the block diagram.

Coercion Dots

Coercion dots appear on block diagram nodes to alert you that you wired two different numeric data types together. The dot means that LabVIEW converted the value passed into the node to a different representation. For example, the Add function expects two double-precision floating-point inputs. If you change one of those inputs to an integer, a coercion dot appears on the Add function.

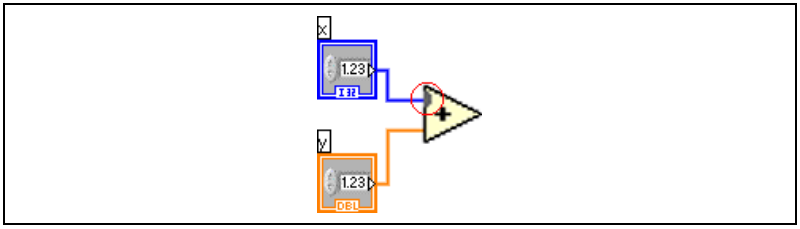


Figure 5-2. Coercion Dot

The block diagram places a coercion dot on the border of a terminal where the conversion takes place to indicate that automatic numeric conversion occurred. Because VIs and functions can have many terminals, a coercion dot can appear inside an icon if you wire through one terminal to another terminal.

Coercion dots also appear on the terminal when you wire any data type to a variant terminal, except when you wire two variants together. Refer to the *Handling Variant Data* section of this chapter for more information about the variant data type.

Coercion dots can cause a VI to use more memory and increase its run time. Try to keep data types consistent in your VIs.

Polymorphic VIs and Functions

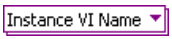
Polymorphic VIs and functions can adjust to input data of different data types. Most LabVIEW structures are polymorphic, as are some VIs and functions.

Polymorphic VIs

Polymorphic VIs accept different data types for a single input or output terminal. A polymorphic VI is a collection of subVIs with the same connector pane patterns. Each subVI is an instance of the polymorphic VI.

For example, the Read Key VI is polymorphic. Its **default value** terminal accepts Boolean; double-precision, floating-point numeric; 32-bit signed integer numeric; path; string; or 32-bit unsigned integer numeric data.

For most polymorphic VIs, the data types you wire to the inputs of the polymorphic VI determine the instance to use. If the polymorphic VI does not contain a subVI compatible with that data type, a broken wire appears. If the data types you wire to the polymorphic VI inputs do not determine the instance to use, you must select the instance manually. If you manually select an instance of a polymorphic VI, the VI no longer behaves as a polymorphic VI because it accepts and returns only the data types of the instance you select.



To select the instance manually, right-click the polymorphic VI, select **Select Type** from the shortcut menu, and select the instance to use. You also can use the Operating tool to click the polymorphic VI selector, shown at left, and select an instance from the shortcut menu. Right-click the polymorphic VI on the block diagram and select **Visible Items»Polymorphic VI Selector** from the shortcut menu to display the selector. To change the polymorphic VI to accept all the handled data types again, right-click the polymorphic VI and select **Select Type»Automatic** from the shortcut menu or use the Operating tool to click the polymorphic VI selector and select **Automatic** from the shortcut menu.

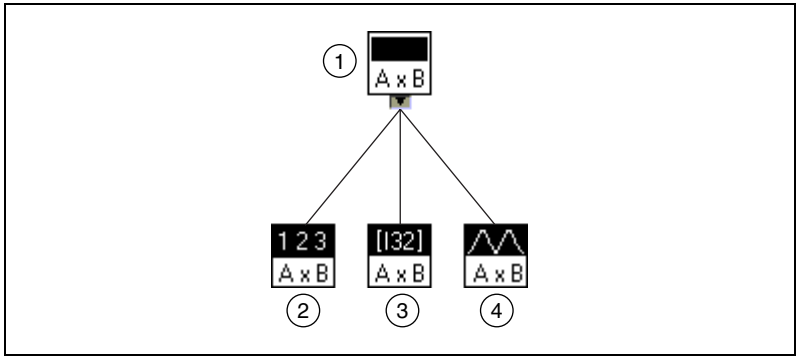
Building Polymorphic VIs

Build polymorphic VIs when you perform the same operation on different data types.

Note You can build and edit polymorphic VIs only in the LabVIEW Professional Development System.

For example, if you want to perform the same mathematical operation on a single-precision floating-point numeric, an array of numerics, or a waveform, you could create three separate VIs—Compute Number, Compute Array, and Compute Waveform. When you need to perform the operation, you place one of these VIs on the block diagram, depending on the data type you use as an input.

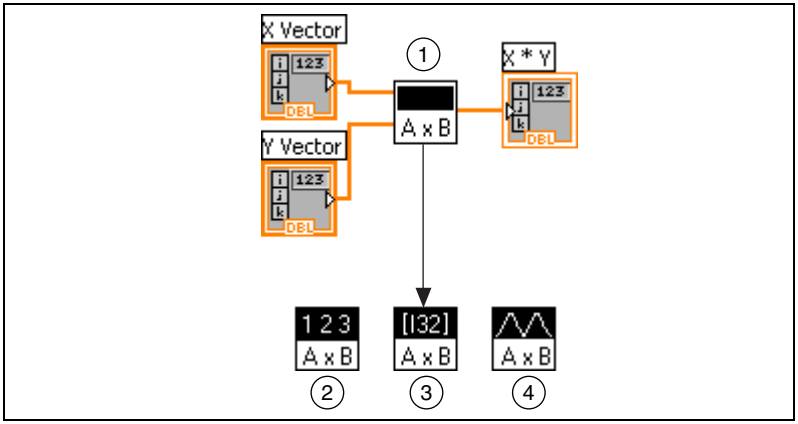
Instead of manually placing a version of the VI on the block diagram, you can create and use a single polymorphic VI. The polymorphic Compute VI contains three instances of the VI, as shown in Figure 5-3.



1	Compute	3	Compute Array
2	Compute Number	4	Compute Waveform

Figure 5-3. Example of a Polymorphic VI

The Compute VI statically links the correct instance of the VI based on the data type you wire to the Compute subVI on the block diagram, as shown in Figure 5-4.



1	Compute	3	Compute Array
2	Compute Number	4	Compute Waveform

Figure 5-4. Polymorphic VI Calling a SubVI

Polymorphic VIs differ from most VIs in that they do not have a block diagram or a front panel.

Consider the following issues when you build polymorphic VIs:

- All VIs you include in the polymorphic VI must have the same connector pane pattern, because the connector pane of the polymorphic VI matches the connector pane of the VIs you use to create the polymorphic VI.
- The inputs and outputs on the connector pane of each instance of the VI must correspond to the inputs and outputs on the connector pane of the polymorphic VI.
- The VIs you use to build polymorphic VIs do not have to consist of the same subVIs and functions.
- Each of the front panels of the VIs do not have to have the same number of objects. However, each front panel must have at least the same number of controls and indicators that make up the connector pane of the polymorphic VI.
- You can create an icon for a polymorphic VI.
- You cannot use polymorphic VIs in other polymorphic VIs.

When you generate complete documentation for a VI that includes a polymorphic subVI, the polymorphic VI and its instances appear in the list of subVIs section of the documentation.

Polymorphic Functions

Functions are polymorphic to varying degrees—none, some, or all of their inputs can be polymorphic. Some function inputs accept numerics or Boolean values. Some accept numerics or strings. Some accept not only scalar numerics, but also arrays of numerics, clusters of numerics, arrays of clusters of numerics, 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 numerics. Refer to Appendix B, *Polymorphic Functions*, for more information about polymorphic functions.

Express VIs

Use the Express VIs for common measurement tasks. Express VIs are function nodes that require minimal wiring because you configure them with dialog boxes. The inputs and outputs for the Express VI depend on how you configure the VI. Express VIs appear on the block diagram as expandable nodes with icons surrounded by a blue field.

Refer to the *Displaying SubVIs and Express VIs as Icons or Expandable Nodes* section of Chapter 7, *Creating VIs and SubVIs*, for more information about expandable nodes. Refer to the *Getting Started with LabVIEW* manual for more information about Express VIs.

Creating Express VIs as SubVIs

You can create a subVI from an Express VI configuration. For example, you might want to save the configuration of the Write LabVIEW Measurement File Express VI for use as a subVI in other VIs you build rather than reconfiguring the Express VI every time. Right-click the Express VI and select **Open Front Panel** from the shortcut menu to create a subVI from an Express VI.

When you create a subVI from an Express VI, the front panel of the subVI appears. You then can edit the VI and save it. To save the values entered in controls, select **Operate»Make Current Values Default** or right-click each control and select **Make Current Value Default** from the shortcut menu. A new subVI displayed as an expandable node replaces the Express VI on the block diagram.

After you create a VI from an Express VI, you cannot convert the subVI back into an Express VI.

Dynamic Data Type

The dynamic data type appears as a dark blue terminal, shown at left. 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 a graph, chart, or numeric indicator.

The dynamic data type is for use with Express VIs. Most other the VIs and functions that ship with LabVIEW do not accept this 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. Refer to the *Converting from Dynamic Data* section of this chapter for more information on converting from dynamic data type.

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 a graph or chart. For example, if you use the DAQ Assistant Express VI to acquire a signal and plot that signal on a graph, the name of the signal appears in the plot legend of the graph, and the x-scale adjusts to display timing information associated with the signal in relative or absolute time based on the attributes of the signal. If you use the Spectral Measurements Express VI to perform an FFT analysis on the signal and plot the resulting value on a graph, the x-scale automatically adjusts to plot the signal in the frequency domain based on the attributes of the signal. Right-click a dynamic data type output terminal of a VI or function on the block diagram and select **Create»Graph Indicator** to display the data in a graph or select **Create»Numeric Indicator** to display the data in a numeric indicator.

Table 5-2 lists indicators that accept the dynamic data type and the type of data the dynamic data type can contain, and describes how indicators handle the data.

Table 5-2. Dynamic Data Type Indicators

Data in Dynamic Data Type	Indicator	Result
Single numeric value	Graph	Plots the single value, includes time stamp and attributes
Single channel		Plots the whole waveform, includes time stamp and attributes
Multiple channels		Plots all the data, includes time stamps and attributes
Single numeric value	Numeric Indicator	Displays the single value
Single channel		Displays the last value of the channel's data
Multiple channels		Displays the last value of the first channel's data
Single numeric value	Boolean Indicator	Displays a TRUE value if numeric value is greater than or equal to .5
Single channel		Displays a TRUE value if the last value of channel's data is greater than or equal to .5
Multiple channels		Displays a TRUE value if the last value of the first channel's data is greater than or equal to .5

Converting from Dynamic Data

Use the Convert from Dynamic Data Express VI to convert the dynamic data type to numeric, 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.

For example, if you acquired a sine wave from a data acquisition device, select the **Single Waveform** option on the **Configure Convert from Dynamic Data** dialog box. Wire the **Waveform** output of the Convert from Dynamic Data Express VI to a function or VI that accepts the waveform data type. If you acquired a collection of temperatures from different channels using a DAQ device, select the **Most recent values from each channel** and **Floating point numbers (double)** options. Then wire the **Array** output of the Convert from Dynamic Data Express VI to a function or VI that accepts a numeric array as an input.

When you wire a dynamic data type to an array indicator, LabVIEW automatically places the Convert from Dynamic Data Express VI on 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.

Converting to Dynamic Data

Use the Convert to Dynamic Data Express VI to convert numeric, Boolean, waveform, and array data types to the dynamic data type for use with Express VIs. When you place the Convert to Dynamic Data Express VI on the block diagram, the **Configure Convert to Dynamic Data** dialog box appears. Use this dialog box to select the kind of data to convert to the dynamic data type.

For example, if you acquire a sine wave using the Analog Input Traditional NI-DAQ VIs and want to use a Signal Analysis Express VI to analyze the signal, select the Single Waveform option on the **Configure Convert to Dynamic Data** dialog box. Then wire the **Dynamic Data Type** output to an Express VI that accepts the dynamic data type as an input.

Handling Variant Data

Variant data do not conform to a specific data type and can contain attributes. LabVIEW represents variant data with the variant data type. The variant data type differs from other data types because it stores the control or indicator name, information about the data type from which you converted, and the data itself, which allows LabVIEW to correctly convert the variant data type to the data type you want. For example, if you convert a string data type to a variant data type, the variant data type stores the text and indicates that the text is a string.

Use the Variant functions to create and manipulate variant data. You can convert any LabVIEW data type to the variant data type to use variant data in other VIs and functions. Several polymorphic functions return the variant data type.

Use the variant data type when it is important to manipulate data independently of data type, such as when you transmit or store data; read and/or write to unknown devices; store information in a stack, queue, or notifier; or perform operations on a heterogeneous set of controls.

You also can use the Flatten to String function to convert a data type to a string data type to represent data independently of type. Flattening data to

strings is useful when you use TCP/IP to transmit data because the protocol understands only strings.

However, using flattened data has limitations because LabVIEW cannot coerce flattened data when the original data type does not match the data type to which you want to convert. Also, attempting to unflatten a flattened integer as an extended-precision floating-point number fails. Refer to the *Flattened Data* section of the *LabVIEW Data Storage* Application Note for more information about flattening and unflattening data.

Another advantage of using the variant data type is the ability to store attributes of the data. An attribute is information about the data that the variant data type stores. For example, if you want to know the time when a piece of data was created, you can store the data as variant data and add an attribute called **Time** to store the time string. The attribute data can be of any type. Use variant attributes when you want to sort the data by a particular attribute, identify the device or application that generated the data, or filter the data for only those variants with a particular attribute.

Numeric Units and Strict Type Checking

You can associate physical units of measure, such as meters or kilometers/second, with any numeric control or indicator that has floating-point representation.

Units for a control appear in a separate owned label, called the unit label. Display the unit label by right-clicking the control and selecting **Visible Items»Unit Label** from the shortcut menu. Right-click the unit label and select **Build Unit String** from the shortcut menu to edit the unit label.

When LabVIEW displays the unit label, you can enter a unit using standard abbreviations such as `m` for meters, `ft` for feet, `s` for seconds, and so on.

Note You cannot use units in Formula Nodes.

Units and Strict Type Checking

When you associate units with an object, you can wire only objects that have compatible units. LabVIEW uses strict type checking to verify that units are compatible. If you wire two objects with incompatible units, LabVIEW returns an error. For example, LabVIEW returns an error if you wire an object with mile as its unit type to an object with liter as its unit type, because a mile is a unit of distance and a liter is a unit of volume.

Figure 5-5 shows wiring objects with compatible units. In this figure, LabVIEW automatically scales the **distance** indicator to display kilometers instead of meters because kilometers is the unit for the indicator.

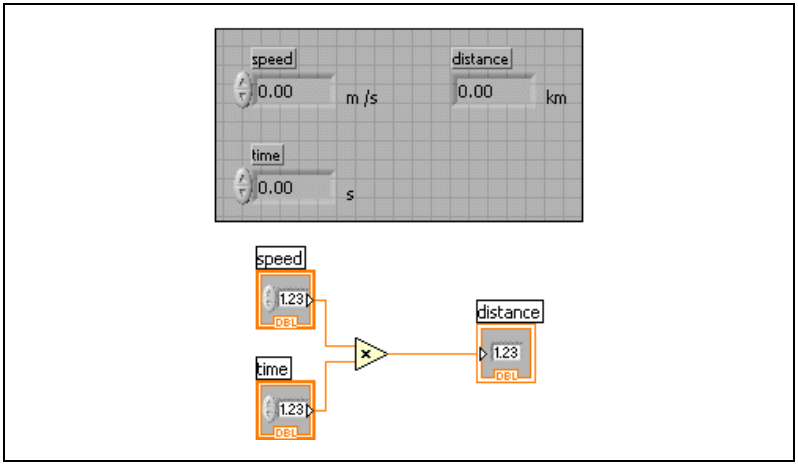


Figure 5-5. Wiring Objects with Compatible Units

An error occurs in Figure 5-6 because **distance** has a unit type of seconds. To correct the error, change seconds to a unit of distance, such as kilometers, as shown in Figure 5-5.

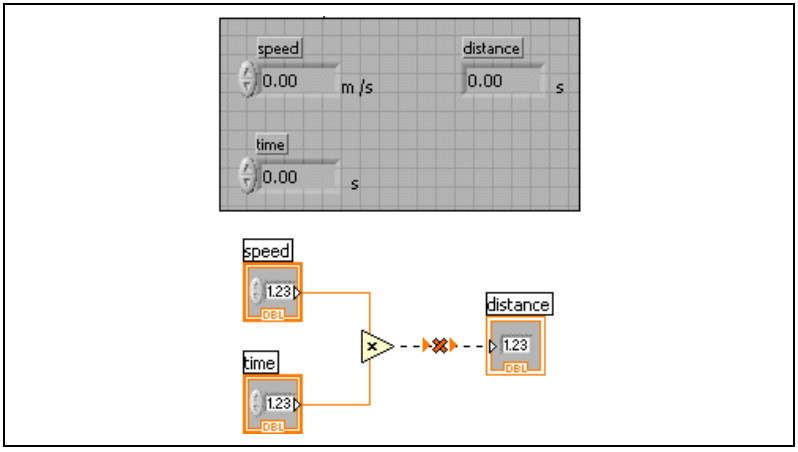


Figure 5-6. Wiring Objects with Incompatible Units Results in Broken Wires

Some VIs and functions are ambiguous with respect to units. You cannot use these VIs and functions with other terminals that have units. For example, the Increment function is ambiguous with respect to units. If you

use distance units, the Increment function cannot tell whether to add one meter, one kilometer, or one foot. Because of this ambiguity, you cannot use the Increment function and other functions that increment or decrement values with data that have associated units.

To avoid ambiguity in this example, use a numeric constant with the proper unit and the Add function to create your own increment unit function, as shown in Figure 5-7.

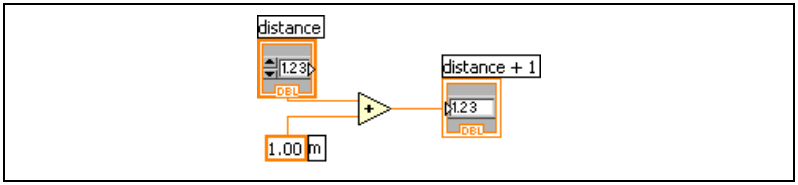


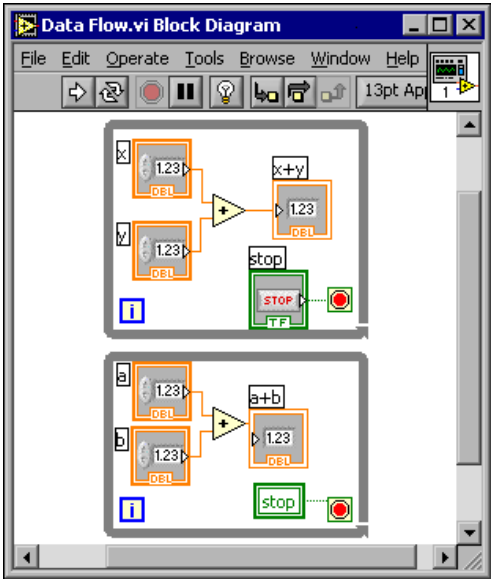
Figure 5-7. Creating an Increment Function with Units

Block Diagram Data Flow

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

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

In LabVIEW, because the flow of data rather than the sequential order of commands determines the execution order of block diagram elements, you can create block diagrams that have simultaneous operations. For example, you can run two While Loops simultaneously and display the results on the front panel.



LabVIEW is a multitasking and multithreaded system, running multiple execution threads and multiple VIs simultaneously. Refer to the *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note for more information about performing tasks simultaneously in LabVIEW.

Data Dependency and Artificial Data Dependency

The control flow model of execution is instruction driven. Dataflow execution is data driven, or data dependent. A node that receives data from another node always executes after the other node completes execution.

Block diagram nodes not connected by wires can execute in any order. Although the *LabVIEW Development Guidelines* manual recommends using a left-to-right and top-to-bottom layout, nodes do not necessarily execute in left-to-right, top-to-bottom order.

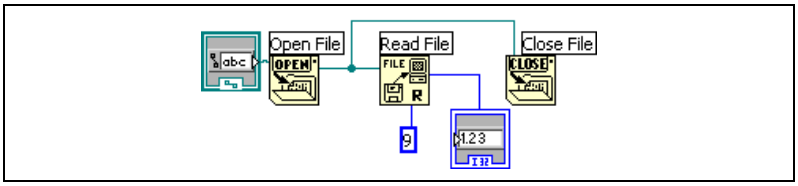
You can use a sequence structure to control execution order when natural data dependency does not exist. Refer to the *Sequence Structures* section of Chapter 8, *Loops and Structures*, for more information about sequence structures. You also can use flow-through parameters to control execution order. Refer to the *Flow-Through Parameters* section of Chapter 14, *File I/O*, for more information about flow-through parameters.

You also can create an artificial data dependency, in which the receiving node does not actually use the data received. Instead, the receiving node uses the arrival of data to trigger its execution. Refer to the Timing Template (data dep) VI in the `examples\general\structs.llb` for an example of using artificial data dependency.

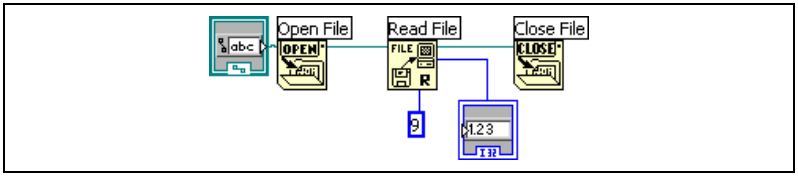
Missing Data Dependencies

Do not assume left-to-right or top-to-bottom execution when no data dependency exists. Make sure you explicitly define the sequence of events when necessary by wiring the dataflow.

In the following example, no dependency exists between the Read File function and the Close File function because the Read File function is not wired to the Close File function. This example might not work as expected because there is no way to determine which function runs first. If the Close File function runs first, the Read File function does not work.



The following block diagram establishes a dependency by wiring an output of the Read File function to the Close File function. The Close File function does not run until it receives the output of the Read File function.



Data Flow and Managing Memory

Dataflow execution makes managing memory easier than the control flow model of execution. In LabVIEW, you do not allocate variables or assign values to them. Instead, you create a block diagram with wires that represent the transition of data.

VIs and functions that generate data automatically allocate the memory for that data. When the VI or function no longer uses the data, LabVIEW deallocates the associated memory. When you add new data to an array or a string, LabVIEW allocates enough memory to manage the new data.

Because LabVIEW automatically handles memory management, you have less control over when memory is allocated or deallocated. If your VI works with large sets of data, you need to understand when memory allocation takes place. Understanding the principles involved can help you write VIs with significantly smaller memory requirements. Minimizing memory usage can help you increase the speed at which VIs run.

Refer to the *LabVIEW Performance and Memory Management* Application Note for more information about memory allocation. Refer to the *Memory and Speed Optimization* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about optimizing memory usage as you develop.

Designing the Block Diagram

Use the following guidelines to design block diagrams:

- Use a left-to-right and top-to-bottom layout. Although the positions of block diagram elements do not determine execution order, avoid wiring from right to left to keep the block diagram organized and easy to understand. Only wires and structures determine execution order.
- Avoid creating a block diagram that occupies more than one or two screens. If a block diagram becomes large and complex, it can be difficult to understand or debug.
- Decide if you can reuse some components of the block diagram in other VIs or if a section of the block diagram fits together as a logical component. If so, divide the block diagram into subVIs that perform specific tasks. Using subVIs helps you manage changes and debug the block diagrams quickly. Refer to the *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about subVIs.
- Use the error handling VIs, functions, and parameters to manage errors on the block diagram. Refer to the *Error Checking and Error Handling* section of Chapter 6, *Running and Debugging VIs*, for more information about handling errors.
- Improve the appearance of the block diagram by wiring efficiently. Poor wire organization might not produce errors, but it can make the block diagram difficult to read and debug or make the VI appear to do things it does not do.

- Avoid wiring under a structure border or between overlapped objects, because LabVIEW might hide some segments of the resulting wire.
- Avoid placing objects on top of wires. Wires connect only those objects you click. Placing a terminal or icon on top of a wire makes it appear as if a connection exists when it does not.
- Use free labels to document code on the block diagram.
- Increase the space between crowded or tightly grouped objects by pressing the <Ctrl> key and use the Positioning tool to click the block diagram workspace. While holding the key combination, drag out a region the size you want to insert.

  (Mac OS) Press the <Option> key. **(Sun)** Press the <Meta> key. **(Linux)** Press the <Alt> key.

  A rectangle marked by a dashed border defines where space will be inserted. Release the key combination to add the space.

Refer to the *Block Diagram Style* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about designing the block diagram.

6

Running and Debugging VIs

To run a VI, you must wire all the subVIs, functions, and structures with the data types the terminals expect. 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.

For more information...

Refer to the *LabVIEW Help* for more information about running and debugging VIs.

Running VIs

Running a VI executes the operation for which you designed the VI. You can run a VI if the **Run** button on the toolbar appears as a solid white arrow, shown at left. 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 changes to darkened arrow, shown at left, to indicate that the VI is running. You cannot edit a VI while the VI runs.

A VI runs when you click the **Run** or **Run Continuously** buttons or the single-stepping buttons on the block diagram toolbar. Clicking the **Run** button runs the VI once. The VI stops when the VI completes its data flow. Clicking the **Run Continuously** button, shown at left, runs the VI continuously until you stop it manually. Clicking the single-stepping buttons runs the VI in incremental steps. Refer to the *Single-Stepping* section of this chapter for more information about using the single-stepping buttons to debug a VI.

Note Avoid using the **Abort Execution** button  to stop a VI. Either let the VI complete its data flow or design a method to stop the VI programmatically. By doing so, the VI is at a known state. For example, place a button on the front panel that stops the VI.

Configuring How a VI Runs

Select **File»VI Properties** and select **Execution** from the **Category** pull-down menu to configure how a VI runs. For example, you can configure a VI to run immediately when it opens or to pause when called as a subVI. You also can configure the VI to run at different priorities. For example, if it is crucial that a VI runs without waiting for another operation to complete, configure the VI to run at time-critical (highest) priority. Refer to the *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note for more information about creating multithreaded VIs. Refer to Chapter 16, *Customizing VIs*, for more information about configuring how VIs run.

Note Incorrectly configuring a VI to run at different priorities can cause unexpected execution. You do not need to configure most VIs to run at different priorities.

Correcting Broken VIs

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button often appears broken, shown at left, when the VI you are creating or editing contains errors. If it is still broken when you finish wiring the block diagram, the VI is broken and cannot run.

Finding Causes for Broken VIs

Click the broken **Run** button or select **Window»Show Error List** to find out why a VI is broken. The **Error list** window lists all the errors. The **VI List** section lists the names of all VIs in memory that have errors. The **errors and warnings** section lists the errors and warnings for the VI you select in the **VI List** section. The **Details** section describes the errors and in some cases recommends how to correct the errors or how to find more information about them. Select **Help** to open an online help file that lists LabVIEW errors and their descriptions.

Click the **Show Error** button or double-click the error description to display the relevant block diagram or front panel and highlight the object that contains the error.

The toolbar includes the **Warning** button, shown at left, if a VI includes a warning and you placed a checkmark in the **Show Warnings** checkbox in the **Error list** window.

Configure LabVIEW to always show warnings in the **Error list** window by selecting **Tools»Options**, selecting **Debugging** from the top pull-down

menu, and placing a checkmark in the **Show warnings in error box by default** checkbox. You can make this change with the **Error list** window open and see the change immediately.

Warnings do not prevent you from running a VI. They are designed to help you avoid potential problems in VIs.

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 *Using Wires to Link Block Diagram Objects* section of Chapter 5, *Building the Block Diagram*, for more information about wiring block diagram objects.
- A required block diagram terminal is unwired. Refer to the *Setting Required, Recommended, and Optional Inputs and Outputs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about required terminals.
- 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 *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about subVIs.

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. Refer to the *Error Handling* section of this chapter for more information about using these parameters.
- To eliminate all VI warnings, select **Windows»Show 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.
- Triple-click the wire with the Operating tool 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 **Browse»Show 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.
- 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
- Comment out a section of the block diagram to determine if the VI performs better without it.
- 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 **Browse»This VI's SubVIs** and **Browse»Unopened SubVIs** to determine if any extra subVIs exist. Also open the **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, shown at left. 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 move from node to node through a VI.

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

If the **error out** cluster reports an error, the error value appears next to **error out** with a red border. If no error occurs, **OK** appears next to **error out** with a green border. Refer to the *Error Clusters* section of this chapter for more information about error clusters.

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 at left, affect execution only in a VI or subVI in single-step mode. Enter single-step mode by clicking the **Step Over** or **Step Into** button on the block diagram toolbar. Move the cursor over the **Step Over**, **Step Into**, 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.

If you single-step through a VI with execution highlighting on, an execution glyph, shown at left, appears on the icons of the subVIs that are currently running.

Probe Tool

Use the Probe tool, shown at left, 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 are incorrect. If data are available, the probe immediately updates during 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.

Types of Probes

You can check intermediate values on a wire when a VI runs by using the 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.

Generic

Use the generic probe to view the data that pass through a wire. Right-click a wire and select **Custom Probe»Generic** 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 pass 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 pass 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 flow 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 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 `examples\general\probes.llb` for an example of using supplied probes.

Custom

Use the Custom Probe Wizard 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 Custom Probe Wizard. Create a probe when you want to have more control over how LabVIEW probes the data that flow 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 Custom Probe Wizard, 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 an unsigned 32-bit 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.

Refer to the *LabVIEW Help* for information about caveats when using and creating custom probes.

Breakpoints

Use the Breakpoint tool, shown at left, to place a breakpoint on a VI, node, or wire on the block diagram and pause execution at that location. When you set a breakpoint on a wire, execution pauses after data pass through the wire. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and uses a marquee to highlight the node or wire 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 values of front panel controls.
- Click the **Pause** button to continue running to the next breakpoint or until the VI finishes running.

LabVIEW saves breakpoints with a VI, but they are active only when you run the VI. You can view all breakpoints by selecting **Browse» Breakpoints**.

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 **Hierarchy** window, which you display by selecting **Browse»Show 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. 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 **Browse»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.

Commenting Out Sections of Block Diagrams

You can run a VI with a section of the block diagram disabled, similar to commenting out a section of code in a text-based programming language. Disable a section of the block diagram to determine if the VI performs better without it.

Place the section you want to disable inside a Case structure and use a Boolean constant to run both cases. Refer to the *Case Structures* section of Chapter 8, *Loops and Structures*, for more information about using Case structures. You also can create a copy of the VI and delete that section of the block diagram from the copy. Discard the version of the VI you decide not to use.

Disabling Debugging Tools

You can disable the debugging tools to reduce memory requirements and to increase performance slightly. Right-click the connector pane and select **VI Properties**. Select **Execution** in the **Category** pull-down menu and remove the checkmark from the **Allow Debugging** checkbox.

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 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. Refer to the *Numeric Conversion* section of Appendix B, *Polymorphic Functions*, for more information about converting numeric values.

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 Comparison function Not A Number/Path/Refnum? to the value you suspect is invalid.

Default Data in Loops

For Loops return default data when the iteration count is zero.

Refer to the *Control and Indicator Data Types* section of Chapter 5, *Building the Block Diagram*, for more information about default values for data types.

For Loops

For Loops produce default data if you wire `0` to the count terminal of the For Loop or if you wire an empty array to the For Loop as an input with auto-indexing enabled. The loop does not execute, and any output tunnel with auto-indexing disabled contains the default value for the tunnel data type. Use shift registers to transfer values through a loop regardless of whether the loop executes.

Refer to the *For Loop and While Loop Structures* section of Chapter 8, *Loops and Structures*, for more information about For Loops, auto-indexing, and shift registers.

Default Data in Arrays

Indexing beyond the bounds of an array produces the default value for the array element parameter. You can use the Array Size function to determine the size of the array. Refer to the *Arrays* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about arrays. Refer to the *Auto-Indexing Loops* section of Chapter 8, *Loops and Structures*, for more information about indexing. You can index beyond the bounds of an array inadvertently by indexing an array past the last element using a While Loop, by supplying too large a value to the **index** input of an Index Array function, or by supplying an empty array to an Index Array function.

Preventing Undefined Data

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 or substitute defined data for the values that the loop creates.

Error Checking and Error Handling

Each error has a numeric code and a corresponding error message. 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 the **Error** dialog box.

Select **File»VI Properties** and select **Execution** from the **Category** pull-down menu to disable automatic error handling for a specific VI. Select **Tools»Options** and select **Block Diagram** from the top pull-down menu to disable automatic error handling for any new VIs you create.

To ignore any error a subVI or function encounters, wire the **error out** output of the subVI or function to the **error in** input of the Clear Errors VI. To disable automatic error handling for a subVI or function, wire its **error out** parameter to the **error in** parameter of another subVI or function or to an **error out** indicator.

Use the LabVIEW error handling VIs, functions, and parameters to manage errors. For example, if LabVIEW encounters an error, you can display the error message in a dialog box. Use error handling in conjunction with the debugging tools to find and manage errors. National Instruments strongly recommends using error handling.

Checking for Errors

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.

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.

Checking for errors in VIs can help you identify the following problems:

- You initialized communications incorrectly or wrote improper data to an external device.
- An external device lost power, is broken, or is working improperly.

- You upgraded the operating system software, which changed the path to a file or the functionality of a VI or library. You might notice a problem in a VI or a system program.

Error Handling

By default, LabVIEW automatically handles errors by suspending execution. You can choose other error handling methods. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

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. Refer to the *Error Clusters* section of this chapter for more information about error clusters.

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

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

Error Clusters

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

- **status** is a Boolean value that reports TRUE if an error occurred. Most VIs, functions, and structures that accept Boolean data also recognize this parameter. For example, you can wire an error cluster to the Boolean inputs of the Stop, Quit LabVIEW, or Select functions. If an error occurs, the error cluster passes a TRUE value to the function.

- **code** is a 32-bit signed integer that identifies the error numerically. A non-zero error code coupled with a **status** of FALSE signals a warning rather than a fatal error.
- **source** is a string that identifies where the error occurred.

Using While Loops for Error Handling

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

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

Using Case Structures for Error Handling

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

7

Creating VIs and SubVIs

After you learn how to build a front panel and block diagram, you can create your own VIs and subVIs, distribute VIs, and build stand-alone applications and shared libraries.

Refer to the *LabVIEW Development Guidelines* manual for more information about planning your project, including information about common development pitfalls and tools you can use to develop your project.

For more information…

Refer to the *LabVIEW Help* for more information about creating and using subVIs, saving VIs, and building stand-alone applications and shared libraries.

Planning and Designing Your Project

Before you develop your own VIs, create a list of tasks your users will need to perform. Determine the user interface components and the number and type of controls and indicators you need for data analysis, displaying analysis results, and so on. Think about and discuss with prospective users or other project team members how and when the user will need to access functions and features. Create sample front panels to show to prospective users or project team members and determine if the front panel helps your users accomplish their tasks. Use this interactive process to refine the user interface as necessary.

Divide your application into logical pieces of manageable size. Begin with a high-level block diagram that includes the main components of your application. For example, the block diagram could include a block for configuration, a block for acquisition, a block for analysis of the acquired data, a block for displaying analysis results, a block for saving the data to disk, and a block to handle errors.

After you design the high-level block diagram, define the inputs and outputs. Then, design the subVIs that make up the main components of the high-level block diagram. Using subVIs makes the high-level block

diagram easy to read, debug, understand, and maintain. You also can create subVIs for common or frequent operations that you can reuse. Test subVIs as you create them. You can create higher level test routines, but catching errors in a small module is easier than testing a hierarchy of several VIs. You might find that the initial design of the high-level block diagram is incomplete. Using subVIs to accomplish low-level tasks makes it easier to modify or reorganize your application. Refer to the *SubVIs* section of this chapter for more information about subVIs.

Select **Help»Find Examples** for examples of block diagrams and subVIs.

Designing Projects with Multiple Developers

If multiple developers work on the same project, define programming responsibilities, interfaces, and coding standards in the beginning to ensure the development process and the application work well together. Refer to Chapter 6, *LabVIEW Style Guide*, of the *LabVIEW Development Guidelines* manual for more information about coding standards.

Keep master copies of the project VIs on a single computer and institute a source code control policy. Consider using the LabVIEW Professional Development System, which includes source code control tools that simplify file sharing. The tools also include a utility to compare VIs and view the changes that were made between versions of VIs. Refer to the *Source Code Control* section of Chapter 2, *Incorporating Quality into the Development Process*, in the *LabVIEW Development Guidelines* manual for more information about using source code control.

VI Templates

The LabVIEW VI templates include the subVIs, functions, structures, and front panel objects you need to get started building common measurement applications. VI templates open as untitled VIs that you must save. Select **File»New** to display the **New** dialog box, which includes the LabVIEW VI templates. You also can display the **New** dialog box by selecting **New** from the **New** pull-down menu of the **LabVIEW** dialog box.

If you use a VI template as a subVI, LabVIEW prompts you to save the template as a VI before you closing the VI.

Creating VI Templates

Create custom VI templates to avoid placing the same components on the front panel or block diagram each time you want to perform a similar operation. Create a custom VI template by building a VI and saving it as a template.

Other Document Types

Select an item under the **Other Document Types** heading in the **Create new** list of the **New** dialog box to start creating global variables, custom controls, run-time menus, polymorphic VIs, or templates for global variables and controls.

Using Built-In VIs and Functions

LabVIEW includes VIs and functions to help you build specific applications, such as data acquisition VIs and functions, VIs that access other VIs, and VIs that communicate with other applications. You can use these VIs as subVIs in your application to reduce development time. Refer to the *SubVIs* section of this chapter for more information about subVIs.

Building Instrument Control and Data Acquisition VIs and Functions

LabVIEW includes hundreds of example VIs you can use and incorporate into VIs that you create. 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.

You can use the built-in VIs and functions to control external instruments, such as oscilloscopes, and to acquire data, such as readings from a thermocouple.

Use the Instrument I/O VIs and functions to control external instruments. To control instruments in LabVIEW, you must have the correct hardware installed, powered on, and operating on your computer. The VIs and functions you use to control instruments depend on the instrumentation communication protocols your hardware supports. Refer to the *LabVIEW Measurements Manual* for more information about building VIs to control instruments.

Use the Data Acquisition VIs and functions to acquire data from DAQ devices. To use these VIs, you must have the NI-DAQ driver software and DAQ hardware installed. Refer to the *LabVIEW Measurements Manual* for

more information about installing the NI-DAQ driver software and DAQ hardware and building VIs to acquire data. After you acquire data, you can use the built-in Analyze, Report Generation, and Mathematics VIs and functions to analyze, generate reports, and perform mathematical operations on that data. Refer to the *LabVIEW Analysis Concepts* manual for more information about analysis and mathematical concepts in LabVIEW.

Building VIs That Access Other VIs

Use the Application Control VIs and functions to control how VIs behave when called as subVIs or run by the user. You can use these VIs and functions to configure multiple VIs at the same time. Also, if you are on a network with other LabVIEW users, you can use these VIs and functions to access and control VIs remotely. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about controlling VIs remotely.

Building VIs That Communicate with Other Applications

Use the File I/O VIs and functions to write data to or read data from other applications, such as Microsoft Excel. You can use these VIs and functions to generate reports or incorporate data from another application in the VI. Refer to Chapter 14, *File I/O*, for more information about passing data to and from other applications.

Use the Communication VIs and functions to transfer LabVIEW data across the Web using a communication protocol such as FTP and to build client-server applications using communication protocols. Refer to Chapter 18, *Networking in LabVIEW*, for more information about communicating with other applications on the network or on the Web.

(Windows) Use the ActiveX functions to add ActiveX objects to VIs or to control ActiveX-enabled applications. Refer to Chapter 19, *Windows Connectivity*, for more information about using ActiveX technology.

SubVIs

After you build a VI and create its icon and connector pane, you can use it in another VI. A VI called from the block diagram of another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages. A subVI node corresponds to a subroutine call in text-based programming languages. The node is not the subVI itself, just as a subroutine call statement in a program is not the subroutine itself. A block

diagram that contains several identical subVI nodes calls the same subVI several times.

The subVI controls and indicators receive data from and return data to the block diagram of the calling VI. Click the **Select a VI** icon on the **Functions** palette, navigate to and double-click a VI, and place the VI on a block diagram to create a subVI call to that VI.

Note Before you can use a VI as a subVI, you must set up a connector pane. Refer to the *Setting up the Connector Pane* section of this chapter for more information about setting up a connector pane.

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.

When LabVIEW calls a subVI, ordinarily the subVI runs without displaying its front panel. If you want a single instance of the subVI to display its front panel when called, right-click the subVI icon and select **SubVI Node Setup** from the shortcut menu. If you want every instance of the subVI to display its front panel when called, select **File»VI Properties**, select **Window Appearance** from the **Category** pull-down menu, and click the **Customize** button.

Watching for Common Operations

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 block diagram in Figure 7-1 contains two identical operations.

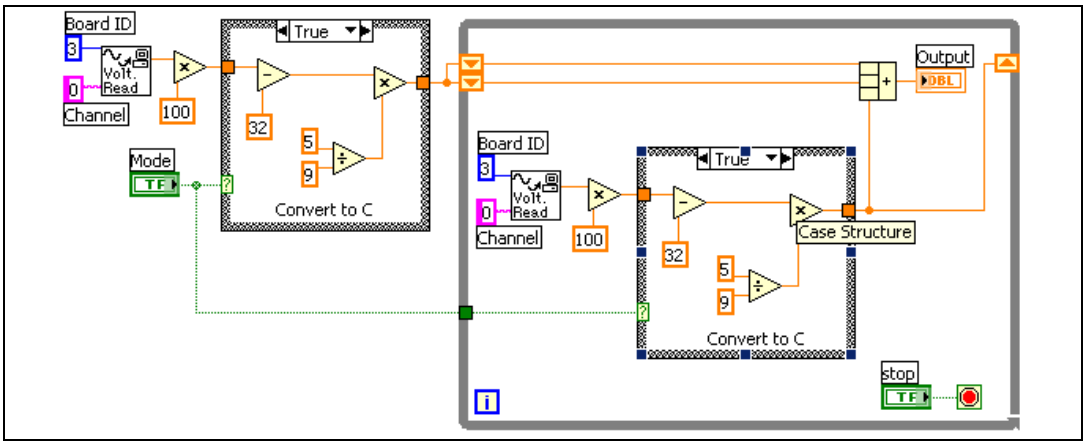


Figure 7-1. Block Diagram with Two Identical Operations

You can create a subVI that performs that operation and call the subVI twice, as shown in Figure 7-2.

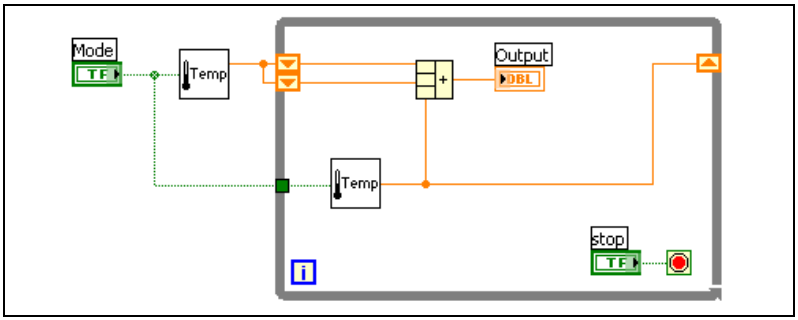


Figure 7-2. Calling a SubVI Twice

You also can reuse the subVI in other VIs. Refer to Chapter 8, *Loops and Structures*, for more information about using loops to combine common operations.

Setting up the Connector Pane

To use a VI as a subVI, you need to build a connector pane, shown at left. The connector pane is a set of terminals 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 defines the inputs and outputs you can wire to the VI so you can use it as a subVI. Refer to the *Icon and Connector Pane* section of Chapter 2, *Introduction to Virtual Instruments*, for more information about connector panes.

Define connections by assigning a front panel control or indicator to each of the connector pane terminals. To define a connector pane, right-click the icon in the upper right corner of the front panel window and select **Show Connector** from the shortcut menu to display the connector pane. The connector pane replaces the icon. Each rectangle on the connector pane represents a terminal. Use the rectangles to assign inputs and outputs. The number of terminals LabVIEW displays on the connector pane depends on the number of controls and indicators on the front panel.

The connector pane has, at most, 28 terminals. 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. Refer to the *Clusters* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about grouping data using clusters.

Note Assigning more than 16 terminals to a VI can reduce readability and usability.

Select a different terminal pattern for a VI by right-clicking the connector pane and selecting **Patterns** from the shortcut menu. 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.

If you create a group of subVIs that you use together often, give the subVIs a consistent connector pane with common inputs in the same location to help you remember where to locate each input. If you create a subVI that produces an output another subVI uses as the input, align the input and output connections to simplify the wiring patterns. Place the **error in** clusters on the lower left corner of the front panel and the **error out** clusters on the lower right corner.

Figure 7-3 shows examples of improperly and properly aligned error clusters.

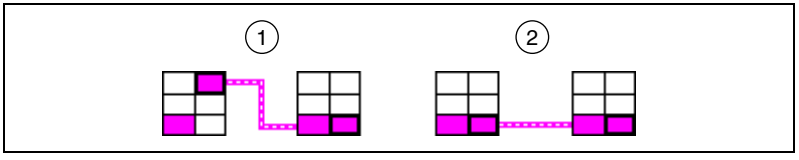


Figure 7-3. Improperly and Properly Aligned Error Clusters

Refer to the *Connector Panes* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about style to use when setting up a connector pane.

Setting Required, Recommended, and Optional Inputs and Outputs

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 in the connector pane and select **This Connection Is** from the shortcut menu. A checkmark indicates the terminal setting. Select **Required**, **Recommended**, or **Optional**.

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.

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.

Creating an Icon

Every VI displays an icon, shown at left, in the upper right corner of the front panel and block diagram windows. An 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.

The default icon contains a number that indicates how many new VIs you have opened since launching LabVIEW. Create custom icons to replace the default icon by right-clicking the icon in the upper right corner of the front

panel or block diagram and selecting **Edit Icon** from the shortcut menu or by double-clicking the icon in the upper right corner of the front panel.

You also can drag a graphic from anywhere in your file system and drop it in the upper right corner of the front panel or block diagram. LabVIEW converts the graphic to a 32×32 pixel icon.

Depending on the type of monitor you use, you can design a separate icon for monochrome, 16-color, and 256-color mode. LabVIEW uses the monochrome icon for printing unless you have a color printer.

Refer to the *Icons* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about designing an icon.

Displaying SubVIs and Express VIs as Icons or Expandable Nodes

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 select **View as Icon** from the shortcut menu to remove the checkmark.

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.

When you resize an expandable subVI or Express VI, the input and output terminals of the subVI or Express VI appear below the icon. Optional terminals appear with gray backgrounds. Recommended or required input or output terminals you do not display appear as input or output arrows in the colored field that surrounds the subVI or Express VI icon. If you wire to an optional terminal when the subVI or Express VI is expanded, then resize the subVI or Express VI so the optional terminal no longer appears in the expanded field, the optional terminal appears as an input or output arrow in the colored field. However, if you unwire the optional terminal, the input or output arrow does not appear.

By default, inputs appear above outputs when you expand the subVI or Express VI. Right-click a terminal in the expandable field and select **Select Input/Output** from the shortcut menu to select an input or output to

display. A line divides inputs from outputs in the shortcut menu. Labels for expandable subVIs and Express VIs appear in the colored field that surrounds the icon. To resize the expandable node so it accommodates the name of each terminal on a single line in the expandable field, right-click the subVI or Express VI and select **Size to Text** from the shortcut menu.

The following expandable subVI displays four of 10 input and output terminals.

Refer to the *Getting Started with LabVIEW* manual for more information about Express VIs.

Creating SubVIs from Sections of a VI

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 select **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 and wires the subVI to the existing wires.

Creating a subVI from a selection is convenient but still requires careful planning to create a logical hierarchy of VIs. Consider which objects to include in the selection and avoid changing the functionality of the resulting VI.

Designing SubVIs

If users do not need to view the front panel of a subVI, you can spend less time on its appearance, including colors and fonts. However, front panel organization is still important because you might need to view the front panel while you debug the VI.

Place the controls and indicators as they appear in the connector pane. Place the controls on the left of the front panel and the indicators on the right. Place the **error in** clusters on the lower left of the front panel and the **error out** clusters on the lower right. Refer to the *Setting up the Connector Pane* section of this chapter for more information about setting up a connector pane.

If a control has a default value, put the default value in parentheses as part of the control name. Also include measurement units, if applicable, in the control name. For example, if a control sets the high limit temperature and has a default value of 75 °F, name the control **high limit temperature (75 degF)**. If you will use the subVI on multiple platforms, avoid using special characters in control names. For example, use **degF** instead of °**F**.

Name Boolean controls so users can determine what the control does in the TRUE state. Use names like **Cancel**, **Reset**, and **Initialize** that describe the action taken.

Viewing the Hierarchy of VIs

The **Hierarchy** window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and global variables. Select **Browse»Show VI Hierarchy** to display the **Hierarchy** window. Use this window to view the subVIs and other nodes that make up the current VI.

As you move the Operating tool over objects in the **Hierarchy** window, LabVIEW displays the name of each VI. You can use the Positioning tool to drag a VI from the **Hierarchy** window to the block diagram to use the VI as a subVI in another VI. You also can select and copy a node or several nodes to the clipboard and paste them on other block diagrams. Double-click a VI in the **Hierarchy** window to display the front panel of that VI.

A VI that contains subVIs has an arrow button on its bottom border. Click this arrow button to show or hide subVIs. A red arrow button appears when all subVIs are hidden. A black arrow button appears when all subVIs are displayed.

Saving VIs

You can save VIs as individual files or you can group several VIs together and save them in a VI library. VI library files end with the extension `.llb`. National Instruments recommends that you save VIs as individual files, organized in directories, especially if multiple developers are working on the same project. Refer to the *Organizing VIs in Directories* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about organizing VIs in directories.

Advantages of Saving VIs as Individual Files

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

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

Advantages of Saving VIs as Libraries

The following list describes reasons to save VIs as libraries:

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

  (Mac OS) Mac OS 9.*x* or earlier limits you to 31 characters for filenames, but has no limit on characters for VI filenames in a library.
- You can transfer a VI library to other platforms more easily than you can transfer multiple individual VIs. You also can ensure that the user receives all the files needed.
- You can slightly reduce the file size of your project because VI libraries are compressed to reduce disk space requirements.
- You can mark VIs in a library as top-level so when you open the library, LabVIEW automatically opens all the top-level VIs in that library.

Note LabVIEW stores many of its built-in VIs and examples in VI libraries to ensure consistent storage locations on all platforms.

If you use VI libraries, consider dividing your application into multiple VI libraries. Put the high-level VIs in one VI library and set up other libraries to contain VIs separated by function. Saving changes to a VI in a library takes longer than saving changes to an individual VI, because the operating system must write the changes to a larger file. Saving changes to a large library also can increase memory requirements and decrease performance. Try to limit the size of each library to approximately 1 MB.

Managing VIs in Libraries

Use the VI Library Manager, available by selecting **Tools»VI Library Manager**, to simplify copying, renaming, and deleting files within VI libraries. You also can use this tool to create new VI libraries and directories and convert VI libraries to and from directories. Creating new VI libraries and directories and converting VI libraries to and from directories is important if you need to manage your VIs with source code control tools.

Before you use the VI Library Manager, close all VIs that might be affected to avoid performing a file operation on a VI already in memory.

(Windows 2000/XP/Me/98) You also can double-click a `.llb` file in Windows Explorer to display the contents of the library so you can open, move, copy, rename, and delete files in the library.

Naming VIs

When you save VIs, use descriptive names. Descriptive names, such as `Temperature Monitor.vi` and `Serial Write & Read.vi`, make it easy to identify a VI and know how you use it. If you use ambiguous names, such as `VI#1.vi`, you might find it difficult to identify VIs, especially if you have saved several VIs.

Consider whether your users will run the VIs on another platform. Avoid using characters that some operating systems reserve for special purposes, such as `\ : / ? * < >` and `#`.

Keep VI names less than 31 characters if your users might run them on Mac OS 9.*x* or earlier.

Saving for a Previous Version

You can save VIs for a previous version of LabVIEW to make upgrading LabVIEW convenient and to help you maintain the VIs in two versions of LabVIEW when necessary. If you upgrade to a new version of LabVIEW, you can go back to the last version of the VIs.

When you save a VI for the previous version, LabVIEW converts not just that VI but all the VIs in its hierarchy, excluding `vi.lib` files.

Often a VI uses functionality not available in the previous version of LabVIEW. In such cases, LabVIEW saves as much of the VI as it can and produces a report of what it cannot convert. The report appears immediately in the **Warnings** dialog box. Click the **OK** button to acknowledge these warnings and close the dialog box. Click the **Save** button to save the warnings to a text file to review later.

Distributing VIs

If you want to distribute VIs to other computers or to other users, consider if you want to include block diagram source code users can edit or if you want to hide or remove the block diagram.

If you want to distribute VIs that include block diagram source code to other LabVIEW users, you can assign password protection to the block diagrams. The block diagram is still available, but users must enter a password to view or edit the block diagram.

If you want to distribute VIs to developers of other programming languages, you can build a stand-alone application or shared library. A stand-alone application or shared library is appropriate when you do not expect your users to edit the VIs. Users can run your application or use the shared library, but they cannot edit or view the block diagrams.

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

Another option for distributing VIs is to remove the block diagram source code so other users cannot edit the VI. Select **File»Save With Options** to save VIs without the block diagrams to reduce the file size and prevent users from changing the source code. If you save a VI without the block diagram, users cannot move the VI to another platform or upgrade the VI to a future version of LabVIEW.

Caution If you save VIs without block diagrams, do *not* overwrite the original versions of the VIs. Save the VIs in different directories or use different names.

Refer to the *Porting and Localizing LabVIEW VIs* Application Note for more information about porting VIs among platforms and localizing VIs.

Building Stand-Alone Applications and Shared Libraries

Select **Tools»Build Application or Shared Library (DLL)** to use the Application Builder to create stand-alone applications and installers or shared libraries (DLLs) for VIs. Use shared libraries if you want to call the VIs in the shared library using text-based programming languages, such as LabWindows™/CVI™, Microsoft Visual C++, and Microsoft Visual Basic.

Shared libraries are useful when you want to share the functionality of the VIs you build with other developers. Using shared libraries provides a way for programming languages other than LabVIEW to access code developed with LabVIEW.

Other developers can run the stand-alone application or use the shared library, but they cannot edit or view the block diagrams.

Note The LabVIEW Professional Development System includes the Application Builder. If you use the LabVIEW Base Package or Full Development System, you can purchase the Application Builder separately by visiting the National Instruments Web site at `ni.com/info` and entering the info code `rdlv21`.Use the tabs in the **Build Application or Shared Library (DLL)** dialog box to configure various settings for the application or shared library you want to build. After you define these settings, save them in a script so you can easily rebuild the application if necessary.

Refer to the *LabVIEW Application Builder User Guide* for more information about installing the Application Builder.

Part II

Building and Editing VIs

This part describes LabVIEW features, VIs, and functions you can use to make your applications operate in specific ways. The chapters in this section describe the usefulness of each LabVIEW feature and outline each class of VIs and functions.

Part II, *Building and Editing VIs*, contains the following chapters:

- Chapter 8, *Loops and Structures*, describes how to use loops and structures on the block diagram to repeat blocks of code and to execute code conditionally or in a specific order.
- Chapter 9, *Event-Driven Programming*, describes how to dynamically and statically register events and how to create and name your own events.
- Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, describes how to use strings, arrays, and clusters to group data.
- Chapter 11, *Local and Global Variables*, describes how to use local and global variables pass information between locations in your application that you cannot connect with a wire.
- Chapter 12, *Graphs and Charts*, describes how to use graphs and charts to display plots of data in a graphical form.
- Chapter 13, *Graphics and Sound VIs*, describes how to display or modify graphics and sound in VIs.
- Chapter 14, *File I/O*, describes how to perform file I/O operations.
- Chapter 15, *Documenting and Printing VIs*, describes how to document and print VIs.
- Chapter 16, *Customizing VIs*, describes how to configure VIs and subVIs to work according to your application needs.

- Chapter 17, *Programmatically Controlling VIs*, describes how to communicate with VIs and other instances of LabVIEW so you can programmatically control VIs and LabVIEW.
- Chapter 18, *Networking in LabVIEW*, describes how to use VIs to communicate, or network, with other processes, including those that run on other applications or on remote computers.
- Chapter 19, *Windows Connectivity*, describes how to provide a public set of objects, commands, and functions that other Windows applications can access.
- Chapter 20, *Calling Code from Text-Based Programming Languages*, describes how to call code from text-based programming languages and use DLLs.
- Chapter 21, *Formulas and Equations*, describes how to use equations in VIs.

8

Loops and Structures

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

Like other nodes, structures have terminals that connect them to other block diagram nodes, execute automatically when input data are available, and supply data to output wires when execution completes.

Each structure has a distinctive, resizable border to enclose the section of the block diagram that executes according to the rules of the structure. The section of the block diagram inside the structure border is called a subdiagram. The terminals that feed data into and out of structures are called tunnels. A tunnel is a connection point on a structure border.

For more information...

Refer to the *LabVIEW Help* for more information about using structures.

Use the following structures located on the **Structures** palette to control how a block diagram executes processes:

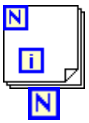
- **For Loop**—Executes a subdiagram a set number of times.
- **While Loop**—Executes a subdiagram until a condition is met.
- **Case structure**—Contains multiple subdiagrams, only one of which executes depending on the input value passed to the structure.
- **Sequence structure**—Contains one or more subdiagrams, which execute in sequential order.
- **Formula Node**—Performs mathematical operations based on numeric input. Refer to the *Formula Nodes* section of Chapter 21, *Formulas and Equations*, for information about using Formula Nodes.
- **Event structure**—Contains one or more subdiagrams that execute depending on how the user interacts with the VI.

Right-click the border of the structure to display the shortcut menu.

For Loop and While Loop Structures

Use the For Loop and the While Loop to control repetitive operations.

For Loops



A For Loop, shown at left, executes a subdiagram a set number of times.

The value in the count terminal (an input terminal), shown at left, indicates how many times to repeat the subdiagram. Set the count explicitly by wiring a value from outside the loop to the left or top side of the count terminal, or set the count implicitly with auto-indexing. Refer to the *Auto-Indexing to Set the For Loop Count* section of this chapter for more information about setting the count implicitly.

The iteration terminal (an output terminal), shown at left, contains the number of completed iterations. The iteration count always starts at zero. During the first iteration, the iteration terminal returns `0`.

Both the count and iteration terminals are 32-bit signed integers. If you wire a floating-point number to the count terminal, LabVIEW rounds it and coerces it to within range. If you wire `0` or a negative number to the count terminal, the loop does not execute and the outputs contain the default data for that data type.

Add shift registers to the For Loop to pass data from the current iteration to the next iteration. Refer to the *Shift Registers and the Feedback Node in Loops* section of this chapter for more information about adding shift registers to a loop.

While Loops

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown at left, executes a subdiagram until a condition is met.

The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value. The default behavior and appearance of the conditional terminal is **Stop if True**, shown at left. When a conditional terminal is **Stop if True**, the While Loop executes its subdiagram until the conditional terminal receives a TRUE value. You can change the behavior and appearance of the conditional terminal by right-clicking the terminal or the border of the While Loop and selecting **Continue if True**, shown at left, from the shortcut menu. When a conditional terminal is **Continue if True**, the While Loop executes its

subdiagram until the conditional terminal receives a FALSE value. You also can use the Operating tool to click the conditional terminal to change the condition.

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**. Refer to the *Error Checking and Error Handling* section of Chapter 6, *Running and Debugging VIs*, for more information about error clusters and error handling.

The iteration terminal (an output terminal), shown at left, contains the number of completed iterations. The iteration count always starts at zero. During the first iteration, the iteration terminal returns `0`.

Add shift registers to the While Loop to pass data from the current iteration to the next iteration. Refer to the *Shift Registers and the Feedback Node in Loops* section of this chapter for more information about adding shift registers to a loop.

Avoiding Infinite While Loops

If you place the terminal of the Boolean control outside the While Loop, as shown in Figure 8-1, and the control is set to FALSE if the conditional terminal is **Stop if True** when the loop starts, you cause an infinite loop. You also cause an infinite loop if the control outside the loop is set to TRUE and the conditional terminal is **Continue if True**.

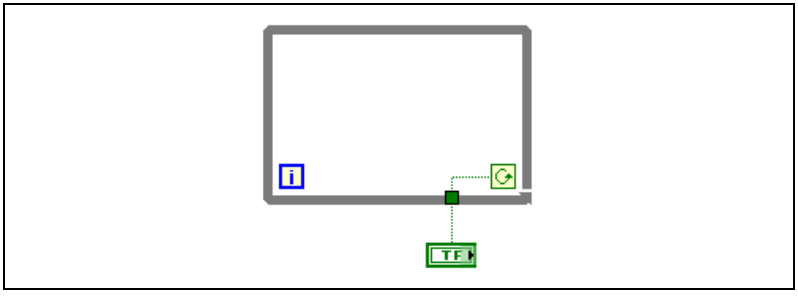


Figure 8-1. Infinite While Loop

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** button on the toolbar.

Auto-Indexing Loops

If you wire an array to a For Loop or While Loop input tunnel, you can read and process every element in that array by enabling auto-indexing. Refer to Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about arrays.

When you wire an array to an input tunnel on the loop border and enable auto-indexing on the input tunnel, elements of that array enter the loop one at a time, starting with the first element. When auto-indexing is disabled, the entire array is passed into the loop. 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.

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.

The loop indexes scalar elements from 1D arrays, 1D arrays from 2D arrays, and so on. The opposite occurs at output tunnels. Scalar elements accumulate sequentially into 1D arrays, 1D arrays accumulate into 2D arrays, and so on.

Auto-Indexing to Set the For Loop Count

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 an element at a time, LabVIEW enables auto-indexing by default for every array you wire to a For Loop. Disable auto-indexing if you do not need to process arrays one element at a time.

If you enable auto-indexing for more than one tunnel or if you wire the count terminal, the count 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 executes 10 times, and the loop indexes only the first 10 elements of the second array. If you plot data from two sources on one graph and you want to plot the first 100 elements, wire 100 to the count terminal. If one of the data sources includes only 50 elements, the loop executes 50 times and indexes only the first 50 elements. Use the Array Size function to determine the size of arrays.

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. For example, if the loop executes 10 times, the output array has 10 elements. If you disable auto-indexing on an output tunnel, only the element from the last iteration of the loop passes to the next node on the block diagram. A bracketed glyph appears on the loop border to indicate that auto-indexing is enabled. The thickness of the wire between the output tunnel and the next node also indicates the loop is using auto-indexing. The wire is thicker when you use auto-indexing because the wire contains an array, instead of a scalar.

Auto-Indexing with While Loops

If you enable auto-indexing for an array entering a While Loop, the While Loop indexes the array the same way a For Loop does. However, the number of iterations a While Loop executes is not limited by the size of the array because the While Loop iterates until a specific condition is met. When a While Loop indexes past the end of the input array, the default value for the array element type passes into the loop. You can prevent the default value from passing into the While Loop by using the Array Size function. The Array Size function indicates how many elements are in the array. Set up the While Loop to stop executing when it has iterated the same number of times as the array size.

Caution Because you cannot determine the size of the output array in advance, enabling auto-indexing for the output of a For Loop is more efficient than with a While Loop. Iterating too many times can cause your system to run out of memory.

Using Loops to Build Arrays

In addition to using loops to read and process elements in an array, you also can use the For Loop and the While Loop to build arrays. Wire the output of a VI or function in the loop to the loop border. If you use a While Loop, right-click the resulting tunnel and select **Enable Indexing** from the shortcut menu. On the For Loop, indexing is enabled by default. The output of the tunnel is an array of every value the VI or function returns after each loop iteration.

Refer to the `examples\general\arrays.llb` for examples of building arrays.

Shift Registers and the Feedback Node in Loops

Use shift registers or the Feedback Node with For Loops or While Loops to transfer values from one loop iteration to the next.

Shift Registers

Use shift registers when you want to pass values from previous iterations through the loop. A shift register appears as a pair of terminals, shown at left, directly opposite each other on the vertical sides of the loop border. The right terminal 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. 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.

Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop. Initializing a shift register resets the value the shift register passes the first time the loop executes when the VI runs. If you do not initialize the register, the loop uses the value written to the register when the loop last executed or the default value for the data type if the loop has never executed.

Use a loop with an uninitialized shift register to run a VI repeatedly so that each time the VI runs, the initial output of the shift register is the last value from the previous execution. Use an uninitialized shift register to preserve state information between subsequent executions of a VI. After the loop executes, the last value stored in the shift register is output from the right terminal.

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.

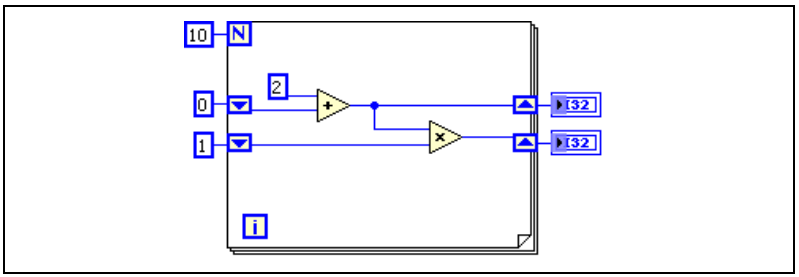


Figure 8-2. Multiple Shift Registers

Stacked Shift Registers

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

Stacked shift registers can only occur on the left side of the loop because the right terminal only transfers the data generated from the current iteration to the next iteration.

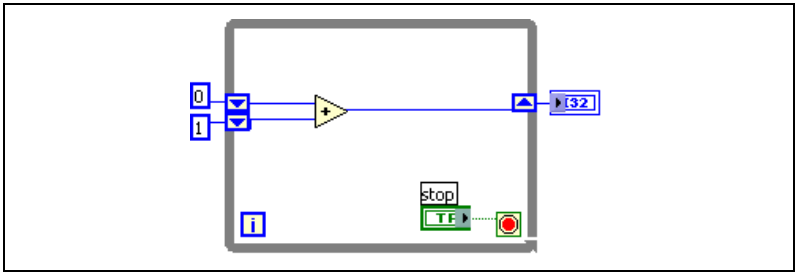


Figure 8-3. Stacked Shift Registers

If you add another element to the left terminal, 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.

Replacing Shift Registers with Tunnels

Replace shift registers with tunnels by right-clicking the shift register and selecting **Replace with Tunnels** from the shortcut menu when you no longer need to transfer values from one loop iteration to the next.

If you replace an output shift register terminal with a tunnel on a For Loop, the wire to any node outside the loop breaks because the For Loop enables indexing by default. Right-click the tunnel and select **Disable Indexing at Source** from the shortcut menu to disable indexing and automatically correct the broken wire. If you want indexing enabled, delete the broken wire and the indicator terminal, right-click the tunnel, and select **Create Indicator**.

Refer to the *Auto-Indexing Loops* section of this chapter for more information about indexing in loops.

Replacing Tunnels with Shift Registers

Replace tunnels with shift registers by right-clicking the tunnel and selecting **Replace with Shift Register** from the shortcut menu when you want to transfer values from one loop iteration to the next. If no tunnel exists on the loop border opposite of the tunnel you right-clicked, LabVIEW automatically creates a pair of shift register terminals. If a tunnel exists on the loop border opposite of the tunnel you right-clicked, LabVIEW replaces the tunnel you right-clicked with a shift register terminal, and the cursor becomes a shift register icon. Click a tunnel on the opposite side of the loop to replace the tunnel with a shift register or click the block diagram to place the shift register on the loop border directly across from the other shift register terminal. If the shift register terminal appears behind a tunnel, the shift register is not wired.

If you convert a tunnel with indexing enabled to a shift register on a While Loop, the wire to any node outside the loop breaks because shift registers cannot auto index. Delete the broken wire, wire the output wire to the right of the shift register to another tunnel, right-click the tunnel, and select **Enable Indexing** from the shortcut menu, and wire the tunnel to the node.

Refer to the *Auto-Indexing Loops* section of this chapter for more information about indexing in loops.

Feedback Node

The Feedback Node, shown at left, appears automatically only in a For Loop or While Loop when you wire the output of a subVI, function, or group of subVIs and functions to the input of that same VI, function, or group. Like a shift register, the Feedback Node stores data when the loop completes an iteration, sends that value to the next iteration of the loop, and transfers any data type. Use the Feedback Node to avoid unnecessarily long wires in loops. The Feedback Node arrow indicates in which direction the data flows along the wire.

You also can select the Feedback Node and place it only inside a For Loop or While Loop. If you place the Feedback Node on the wire before you branch the wire that connects the data to the tunnel, the Feedback Node passes each value to the tunnel. If you place the Feedback Node on the wire after you branch the wire that connects data to the tunnel, the Feedback Node passes each value back to the input of the VI or function and then passes the last value to the tunnel.

For example, the For Loop in Figure 8-4 iterates 10 times. The Feedback Node passes the value from the previous iteration of the loop to the tunnel before it passes the value to the input of the Add function. The value in the tunnel is always the value from the previous iteration. On the last iteration of the loop, the Feedback Node holds the last value, which in this case is 45, but does not pass that value to the tunnel or to the numeric indicator. When the VI finishes running, the value in the numeric indicator is 36, which is the value from the previous, not the last, iteration of the loop.

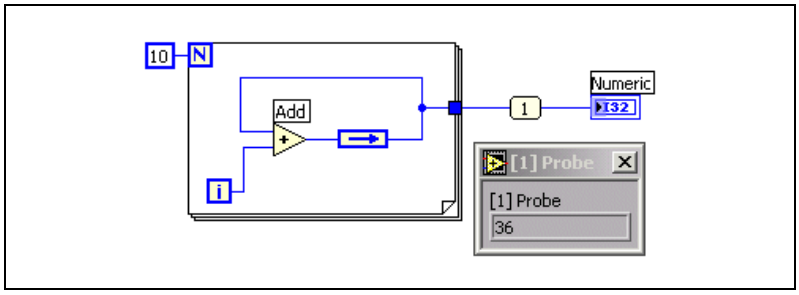


Figure 8-4. Passing the Next to Last Value Out of the For Loop

The For Loop in Figure 8-5 also iterates 10 times. However, the Feedback Node passes the value from the previous iteration only to the input of the Add function each time the loop iterates. On the last iteration of the loop, the Feedback Node passes the value from the previous iteration (36) to the input of the Add function. The Add function adds the value the iteration terminal generates (9) to the value the Feedback Node passes (36) and sends the result to the tunnel. When the VI finishes running, the value in the numeric indicator is **45**.

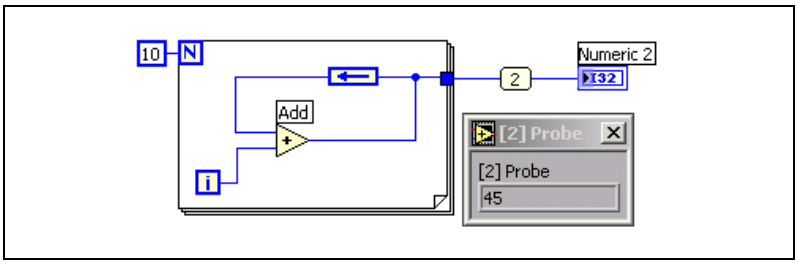


Figure 8-5. Passing the Last Value Out of the For Loop

Initializing Feedback Nodes

Right-click the Feedback Node and select **Initializer Terminal** from the shortcut menu to add the Initializer terminal to the loop border to initialize the loop. When you select the Feedback Node or if you convert an initialized shift register to a Feedback Node, the loop appears with an Initializer terminal automatically. Initializing a Feedback Node resets the initial value the Feedback Node passes the first time the loop executes when the VI runs. If you do not initialize the Feedback Node, the Feedback Node passes the last value written to the node or the default value for the data type if the loop has never executed. If you leave the input of the Initializer terminal unwired, each time the VI runs, the initial input of the Feedback Node is the last value from the previous execution.

Replacing Shift Registers with a Feedback Node

Replace a shift register with a Feedback Node by right-clicking the shift register and selecting **Replace with Feedback Node** from the shortcut menu. Replace a Feedback Node with shift registers by right-clicking the Feedback Node and selecting **Replace with Shift Register** from the shortcut menu.

Controlling Timing

You might want to control the speed at which a process executes, such as the speed at which data are plotted to a chart. You can use a wait function in the loop to wait an amount of time in milliseconds before the loop re-executes.

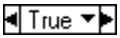
Refer to the *Memory and Speed Optimization* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about using a wait function in a loop to optimize memory usage.

Case and Sequence Structures

Case, Stacked Sequence, Flat Sequence, and Event structures contain multiple subdiagrams. A Case structure executes one subdiagram depending on the input value passed to the structure. A Stacked Sequence and a Flat Sequence structure execute all their subdiagrams in sequential order. An Event structure executes its subdiagrams depending on how the user interacts with the VI.

Case Structures

A Case structure, shown at left, 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 case statements or `if...then...else` statements in text-based programming languages.



The case selector label at the top of the Case structure, shown at left, 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, shown at left, 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.

Specify a default case for the Case structure to handle out-of-range values. Otherwise, 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 valid integer value.

Case Selector Values and Data Types

You can enter a single value or lists and ranges of values in the case selector label. For lists, use commas to separate values. For numeric ranges, specify a range as `10..20`, meaning all numbers from 10 to 20 inclusively. You also can use open-ended ranges. For example, `..100` represents all

numbers less than or equal to 100, and `100..` represents all numbers greater than or equal to 100. You also can combine lists and ranges, for example `..5, 6, 7..10, 12, 13, 14`. When you enter values that contain overlapping ranges in the same case selector label, the Case structure redisplays the label in a more compact form. The previous example redisplays as **`..10, 12..14`**. For string ranges, a range of `a..c` includes all of `a` and `b`, but not `c`. A range of `a..c,c` includes the ending value of `c`.

When you enter string and enumerated values in a case selector label, the values display in quotation marks, for example `"red"`, `"green"`, and `"blue"`. However, you do not need to type the quotation marks when you enter the values unless the string or enumerated value contains a comma or range symbol (`","` or `".."`). In a string value, use special backslash codes for non-alphanumeric characters, such as `\r` for a carriage return, `\n` for a line feed, and `\t` for a tab. Refer to the *LabVIEW Help* for a list of these backslash codes.

If you change the data type of the wire connected to the selector terminal of a Case structure, the Case structure automatically converts the case selector values to the new data type when possible. If you convert a numeric value, for example 19, to a string, the string value is `"19"`. If you convert a string to a numeric value, LabVIEW converts only those string values that represent a number. The other values remain strings. If you convert a number to a Boolean value, LabVIEW converts `0` to FALSE and `1` to TRUE, and all other numeric values become strings.

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

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 each output tunnel for each case. When you create an output tunnel in one case, tunnels appear at the same position on the border in all the other cases. If at least one output tunnel is not wired, all output tunnels on the structure appear as white squares. You can define

a different data source for the same output tunnel in each case, but the data types must be compatible for each case. 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.

Using Case Structures for Error Handling

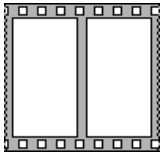
When you wire an error cluster to the selector terminal of a Case structure, the case selector label displays two cases, `Error` and `No Error`, and the border of the Case structure changes color—red for `Error` and green for `No Error`. The Case structure executes the appropriate case subdiagram based on the error state. Refer to the *Error Handling* section of Chapter 6, *Running and Debugging VIs*, for more information about handling errors.

Sequence Structures

A sequence structure contains one or more subdiagrams, or frames, that execute in sequential order. Sequence structures are not used commonly in LabVIEW. Refer the *Using Sequence Structures* and the *Avoiding Overusing Sequence Structures* sections of this chapter for more information on when to use sequence structures.

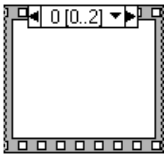
There are two types of sequence structures, the Flat Sequence structure and the Stacked Sequence structure.

Flat Sequence Structure

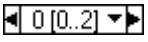


The Flat Sequence structure, shown at left, displays all the frames at once and executes the frames from left to right until the last frame executes. Use the Flat Sequence structure to avoid using sequence locals and to better document the block diagram. When you add or delete frames in a Flat Sequence structure, the structure resizes automatically. To rearrange the frames in a Flat Sequence structure, cut and paste from one frame to another.

Stacked Sequence Structure



The Stacked Sequence structure, shown at left, stacks each frame so you see only one frame at a time and executes frame 0, then frame 1, and so on until the last frame executes. The Stacked Sequence structure returns data only after the last frame executes. Use the Stacked Sequence structure if you want to conserve space on the block diagram.



The sequence selector identifier, shown at left, at the top of the Stacked Sequence structure contains the current frame number and range of frames.

Use the sequence selector identifier to navigate through the available frames and rearrange frames. The frame label in a Stacked Sequence structure is similar to the case selector label of the Case structure. The frame label contains the frame number in the center and decrement and increment arrows on each side. Click the decrement and increment arrows to scroll through the available frames. You also can click the down arrow next to the frame number and select a frame from the pull-down menu. Right-click the border of a frame, select **Make This Frame**, and select a frame number from the shortcut menu to rearrange the order of a Stacked Sequence structure.

Unlike the case selector label, you cannot enter values in the frame label. When you add, delete, or rearrange frames in a Stacked Sequence structure, LabVIEW automatically adjusts the numbers in the frame labels.

Using Sequence Structures

Use the sequence structures to control the execution order when natural data dependency does not exist. A node that receives data from another node depends on the other node for data and always executes after the other node completes execution.

Within each frame of a sequence structure, as in the rest of the block diagram, data dependency determines the execution order of nodes. Refer to the *Data Dependency and Artificial Data Dependency* section of Chapter 5, *Building the Block Diagram*, for more information about data dependency.

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

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

Refer to the `examples\general\structs.llb` for examples of using sequence structures.

Avoiding Overusing Sequence Structures

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

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

Also, do not use sequence structures to update an indicator from multiple frames of the sequence structure. For example, a VI used in a test application might have a **Status** indicator that displays the name of the current test in progress. If each test is a subVI called from a different frame, you cannot update the indicator from each frame, as shown by the broken wire in the Stacked Sequence structure in Figure 8-6.

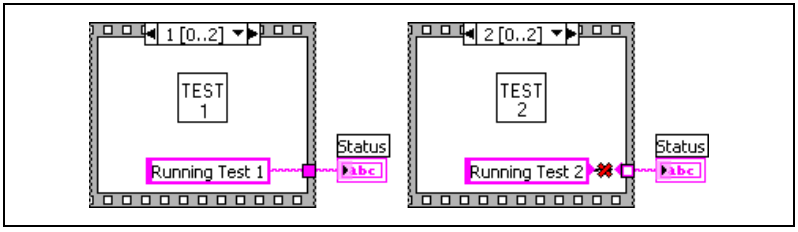


Figure 8-6. Updating an Indicator from Different Frames in a Stacked Sequence Structure

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

Instead, use a Case structure and a While Loop, as shown in Figure 8-7.

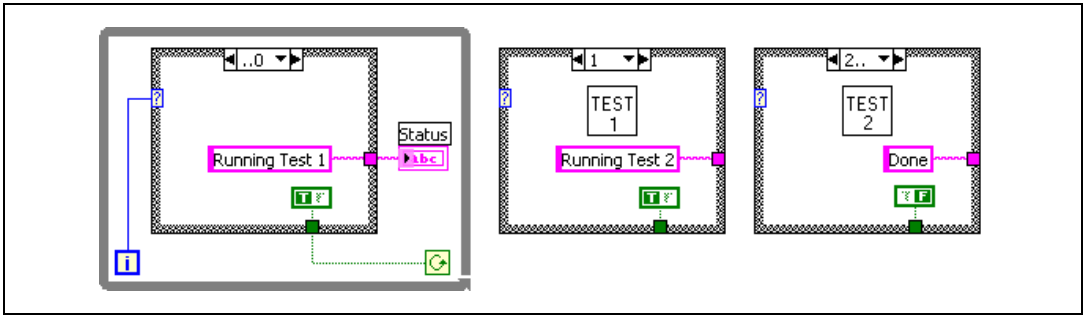


Figure 8-7. Updating an Indicator from Different Cases in a Case Structure

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

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

Replacing Sequence Structures

Right-click a Flat Sequence structure and select **Replace with Stacked Sequence Structure** from the shortcut menu to convert a Flat Sequence structure to a Stacked Sequence structure. Right-click a Stacked Sequence structure and select **Replace»Replace with Flat Sequence** from the shortcut menu to convert a Stacked Sequence structure to a Flat Sequence structure.

9

Event-Driven Programming

LabVIEW is a dataflow programming environment where the flow of data determines the execution order of block diagram elements. Event-driven programming features extend the LabVIEW dataflow environment to allow the user's direct interaction with the front panel and other asynchronous activity to further influence block diagram execution.

Note Event-driven programming features are available only in the LabVIEW Full and Professional Development Systems. You can run a VI built with these features in the LabVIEW Base Package, but you cannot reconfigure the event-handling components.

For more information…

Refer to the *LabVIEW Help* for more information about using events in your application.

What Are Events?

An event is an asynchronous notification that something has occurred. Events can originate from the user interface, external I/O, or other parts of the program. User interface events include mouse clicks, key presses, and so on. External I/O events include hardware timers or triggers that signal when data acquisition completes or when an error condition occurs. Other types of events can be generated programmatically and used to communicate with different parts of the program. LabVIEW supports user interface and programmatically generated events but does not support external I/O events.

In an event-driven program, events that occur in the system directly influence the execution flow. In contrast, a procedural program executes in a pre-determined, sequential order. Event-driven programs usually include a loop that waits for an event to occur, executes code to respond to the event, and reiterates to wait for the next event. How the program responds to each event depends on the code written for that specific event. The order in which an event-driven program executes depends on which events occur

and on the order in which they occur. Some sections of the program might execute frequently because the events they handle occur frequently, and other sections of the program might not execute at all because the events never occur.

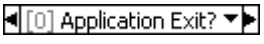
Why Use Events?

Use user interface events in LabVIEW to synchronize user actions on the front panel with block diagram execution. Events allow you to execute a specific event-handling case each time a user performs a specific action. Without events, the block diagram must poll the state of front panel objects in a loop, checking to see if any change has occurred. Polling the front panel requires a significant amount of CPU time and can fail to detect changes if they occur too quickly. By using events to respond to specific user actions, you eliminate the need to poll the front panel to determine which actions the user performed. Instead, LabVIEW actively notifies the block diagram each time an interaction you specified occurs. Using events reduces the CPU requirements of the program, simplifies the block diagram code, and guarantees that the block diagram can respond to all interactions the user makes.

Use programmatically generated events to communicate among different parts of the program that have no dataflow dependency. Programmatically generated events have many of the same advantages as user interface events and can share the same event handling code, making it easy to implement advanced architectures, such as queued state machines using events.

Event Structure Components

Use the Event structure, shown at left, to handle events in a VI. The Event structure works like a Case structure with a built-in Wait On Notification function. The Event structure can have multiple cases, each of which is a separate event-handling routine. You can configure each case to handle one or more events, but only one of these events can occur at a time. When the Event structure executes, it waits until one of the configured events occur, then executes the corresponding case for that event. The Event structure completes execution after handling exactly one event. It does not implicitly loop to handle multiple events. Like a Wait on Notification function, the Event structure can time out while waiting for notification of an event. When this occurs, a specific Timeout case executes.



The event selector label at the top of the Event structure, shown at left, indicates which events cause the currently displayed case to execute. View other event cases by clicking the down arrow next to the case name and selecting another case from the shortcut menu.

(Windows) You also can move the cursor over the selector label and press the <Ctrl> key while moving the mouse wheel. **(UNIX)** Press the <Meta> key.

The Timeout terminal at the top left corner of the Event structure, shown at left, specifies the number of milliseconds to wait for an event before timing out. The default is –1, which specifies to wait indefinitely for an event to occur. If you wire a value to the Timeout terminal, you must provide a Timeout case.

The Event Data Node, shown at left, behaves similarly to the Unbundle By Name function. This node is attached to the inside left border of each event case. The node identifies the data LabVIEW provides when an event occurs. You can resize this node vertically to add more data items, and you can set each data item in the node to access any event data element. The node provides different data elements in each case of the Event structure depending on which event(s) you configure that case to handle. If you configure a single case to handle multiple events, the Event Data Node provides only the event data elements that are common to all the events configured for that case.

The Event Filter Node, shown at left, is similar to the Event Data Node. This node is attached to the inside right border of filter event cases. The node identifies the subset of data available in the Event Data Node that the event case can modify. The node displays different data depending on which event(s) you configure that case to handle. By default, these items are inplace to the corresponding data items in the Event Data Node. If you do not wire a value to a data item of an Event Filter Node, that data item remains unchanged. Refer to the *Notify and Filter Events* section of this chapter for more information about filter events.

The dynamic event terminals, shown at left, are available by right-clicking the Event structure and selecting **Show Dynamic Event Terminals** from the shortcut menu. These terminals are used only for dynamic event registration. Refer to the *Dynamic Event Registration* and *Modifying Registration Dynamically* sections of this chapter for more information about using these terminals.

Note Like a Case structure, the Event structure supports tunnels. However, by default you do not have to wire Event structure output tunnels in every case. All unwired tunnels use the default value for the tunnel data type. Right-click a tunnel and deselect **Use Default If Unwired** from the shortcut menu to revert to the default Case structure behavior where tunnels must be wired in all cases.

Refer to the *LabVIEW Help* for information about the default values for data types.

Notify and Filter Events

There are two types of user interface events—notify and filter.

Notify events are an indication that a user action has already occurred, such as when the user has changed the value of a control. Use notify events to respond to an event after it has occurred and LabVIEW has processed it. You can configure any number of Event structures to respond to the same notify event on a specific object. When the event occurs, LabVIEW sends a copy of the event to each Event structure configured to handle the event in parallel.

Filter events inform you that the user has performed an action before LabVIEW processes it, which allows you to customize how the program responds to interactions with the user interface. Use filter events to participate in the handling of the event, possibly overriding the default behavior for the event. In an Event structure case for a filter event, you can validate or change the event data before LabVIEW finishes processing it, or you can discard the event entirely to prevent the change from affecting the VI. For example, you can configure an Event structure to discard the Panel Close? event, preventing the user from interactively closing the front panel of the VI. Filter events have names that end with a question mark, such as Panel Close?, to help you distinguish them from notify events. Most filter events have an associated notify event of the same name, but without the question mark, which LabVIEW generates after the filter event if no event case discarded the event.

As with notify events, you can configure any number of Event structures to respond to the same filter event on a specific object. However, LabVIEW sends filter events sequentially to each Event structure configured for the event. The order in which LabVIEW sends the event to each Event structure depends on the order in which the events were registered. Refer to the *Using Events in LabVIEW* section of this chapter for more information about event registration. Each Event structure must complete its event case for the event before LabVIEW can notify the next Event structure. If an Event structure case changes any of the event data, LabVIEW passes the changed data to subsequent Event structures in the chain. If an Event structure in the chain discards the event, LabVIEW does not pass the event to any Event structures remaining in the chain. LabVIEW completes processing the user action which triggered the event only after all configured Event structures handle the event without discarding it.

Note National Instruments recommends you use filter events only when you want to take part in the handling of the user action, either by discarding the event or by modifying the event data. If you only want to know that the user performed a particular action, use notify events.

Event structure cases that handle filter events have an Event Filter Node as described in the *Event Structure Components* section of this chapter. You can change the event data by wiring new values to these terminals. If you do not wire a data item, that item remains unchanged. You can completely discard an event by wiring a TRUE value to the **Discard?** terminal.

Note A single case in the Event structure cannot handle both notify and filter events. A case can handle multiple notify events but can handle multiple filter events only if the event data items are identical for all events.

Using Events in LabVIEW

LabVIEW can generate many different events. To avoid generating unwanted events, use event registration to specify which events you want LabVIEW to notify you about. LabVIEW supports two models for event registration—static and dynamic.

Static registration allows you to specify which events on the front panel of a VI you want to handle in each Event structure case on the block diagram of that VI. LabVIEW registers these events automatically when the VI runs. Each event is associated with a control on the front panel of the VI, the front panel window of the VI as a whole, or the LabVIEW application. You cannot statically configure an Event structure to handle events for the front panel of a different VI. Configuration is static because you cannot change at run time which events the Event structure handles. Refer to the *Static Event Registration* section of this chapter for more information about using static registration.

Dynamic event registration avoids the limitations of static registration by integrating event registration with the VI Server, which allows you to use Application, VI, and Control references to specify at run time the objects for which you want to generate events. Dynamic registration provides more flexibility in controlling what events LabVIEW generates and when it generates them. However, dynamic registration is more complex than static registration because it requires using VI Server references with block diagram functions to explicitly register and unregister for events rather than handling registration automatically using the information you configured in the Event structure. Refer to the *Dynamic Event Registration* section of this chapter for more information about using dynamic registration.

Note In general, LabVIEW generates user interface events only as a result of direct user interaction with the active front panel. LabVIEW does not generate events, such as Value Change, when you use the VI Server, global variables, local variables, DataSocket, and so on. Refer to the Value (Signaling) property in the *LabVIEW Help* for information about generating a Value Change event programmatically. In many cases, you can use programmatically generated events instead of queues and notifiers.

The event data provided by a LabVIEW event always include a time stamp, an enumeration that indicates which event occurred, and a VI Server reference to the object that triggered the event. The time stamp is a millisecond counter you can use to compute the time elapsed between two events or to determine the order of occurrence. The reference to the object that generated the event is strictly typed to the VI Server class of that object. Events are grouped into classes according to what type of object generates the event, such as Application, VI, or Control. If a single case handles multiple events for objects of different VI Server classes, the reference type is the common parent class of all objects. For example, if you configure a single case in the Event structure to handle events for a digital numeric control and a color ramp control, the type of the control reference of the event source is Numeric because the digital numeric and color ramp controls are in the Numeric class. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using VI Server classes.

Note If you register for the same event on both the VI and Control class, LabVIEW generates the VI event first. LabVIEW generates Control events for container objects, such as clusters, before it generates events for the objects they contain. If the Event structure case for a VI event or for a Control event on a container object discards the event, LabVIEW does not generate further events.

Each Event structure and Register For Events node on the block diagram owns a queue that LabVIEW uses to store events. When an event occurs, LabVIEW places a copy of the event into each queue registered for that event. An Event structure handles all events in its queue and the events in the queues of any Register For Events nodes that you wired to the dynamic event terminals of the Event structure. LabVIEW uses these queues to ensure that events are reliably delivered to each registered Event structure in the order the events occur.

By default, LabVIEW locks the front panel that contains the object that generates an event until all Event structures finish handling the event. While the front panel is locked, LabVIEW does not process front panel activity but places those interactions in a buffer and handles them when the front panel is unlocked. Front panel locking does not affect certain actions,

such as moving the window, interacting with the scrollbars, and clicking the **Abort** button. You can disable front panel locking for notify events by removing the checkmark from the option in the **Edit Events** dialog box. Front panel locking must be enabled for filter events to ensure the internal state of LabVIEW is not altered before it has an opportunity to completely process the pending event.

LabVIEW can generate events even when no Event structure is waiting to handle them. Because the Event structure handles only one event each time it executes, place the Event structure in a While Loop to ensure that an Event structure can handle all events that occur.

Caution If no Event structure executes to handle an event and front panel locking is enabled, the user interface of the VI becomes unresponsive. If this occurs, click the **Abort** button to stop the VI. You can disable panel locking by right-clicking the Event structure and removing the checkmark from the **Lock front panel until the event case for this event completes** checkbox in the **Edit Events** dialog box. You cannot turn off front panel locking for filter events.

Refer to the *LabVIEW Help* for information about caveats when using Event structures and for information about available events.

Static Event Registration

Static event registration is available only for user interface events. Use the **Edit Events** dialog box to configure an Event structure to handle a statically registered event. Select the event source, which can be the application, the VI, or an individual control. Select a specific event the event source can generate, such as Panel Resize, Value Change, and so on. Edit the case to handle the event data according to the application requirements. Refer to the *LabVIEW Help* for more information about the **Edit Events** dialog box and how to register events statically.

LabVIEW statically registers events automatically and transparently when you run a VI that contains an Event structure. LabVIEW generates events for a VI only while that VI is running or when another running VI calls the VI as a subVI.

When you run a VI, LabVIEW sets that top-level VI and the hierarchy of subVIs the VI calls on its block diagram to an execution state called reserved. You cannot edit a VI or click the **Run** button while the VI is in the reserved state because the VI can be called as a subVI at any time while its parent VI runs. When LabVIEW sets a VI to the reserved state, it automatically registers the events you statically configured in all Event

structures on the block diagram of that VI. When the top-level VI finishes running, LabVIEW sets it and its subVI hierarchy to the idle execution state and automatically unregisters the events.

Refer to the `examples\general\uievents.llb` for examples of using static event registration.

Dynamic Event Registration

Dynamic event registration gives you complete control over which events LabVIEW generates and when it generates them. Use dynamic registration if you want event generation to occur during only part of the application or if you want to change which VIs or controls generate events while your application is running. With dynamic registration, you can handle events in a subVI rather than only in the VI where the events are generated.

Handling dynamically registered events involves the following four main steps:

1. Obtain VI Server references to the objects for which you want to handle events.
2. Register for events on the objects by wiring the VI Server references to the Register For Events node.
3. Enclose the Event structure in a While Loop to handle events on the objects until a terminating condition occurs.
4. Use the Unregister For Events function to stop event generation.

To dynamically register for events on an object, first obtain a VI Server reference to the object. Use the Open Application Reference and Open VI Reference functions to obtain application and VI references. To obtain a reference for a control, use a Property Node to query a VI for its controls or right-click the control and select **Create»Reference** from the shortcut menu to create a control reference constant. Refer to the *Application and VI References* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using VI Server references.

Use the Register For Events function to dynamically register for events. You can resize the Register For Events node to show one or more event source inputs. Wire an application, VI, or control reference to each event source input. Right-click each input and select which event you want to register for from the **Events** shortcut menu. The events from which you can select depend on the VI Server reference type you wire to the event source input. The events on the **Events** shortcut menu are the same events you see in the **Edit Events** dialog box when you statically register events. When the

Register For Events node executes, LabVIEW registers for the specified event on the object associated with each event source input reference. Once you register events, LabVIEW queues events as they occur until an Event structure handles them. Actions that occur before the node executes do not generate events unless another object previously registered for them.

Note Unlike a Property Node, the Register For Events node does not require you to wire the top left input. Use this input only when you want to modify an existing registration. Refer to the *Modifying Registration Dynamically* section of this chapter for more information about event re-registration.

The dynamic event terminals, available by right-clicking the Event structure and selecting **Show Dynamic Event Terminals** from the shortcut menu, behave similarly to shift registers. The left terminal accepts an event registration refnum or a cluster of event registration refnums. If you do not wire the inside right terminal, it carries the same data as the left terminal. However, you can wire the event registration refnum or cluster of event registration refnums to the inside right terminal through a Register For Events node and modify the event registration dynamically. Refer to the *Modifying Registration Dynamically* section of this chapter for more information about using dynamic event terminals.

The output of the Register For Events node is an event registration refnum, which is a strict data type LabVIEW uses to propagate information about registered events to the Event structure. You can view the registered events in the **Context Help** window by idling the cursor over the event registration refnum. After you configure the Register For Events node, wire the **event registration refnum** output of the Register For Events node to the dynamic event terminal on the left side of an Event structure and configure the Event structure to handle the registered events. Avoid branching an event registration refnum wire because that allows multiple Event structures to pull events from one queue and introduces a race condition that can cause unpredictable behavior.

To configure an Event structure to handle a dynamically registered event, use the **Edit Events** dialog box. The **Event Sources** section of the dialog box contains a **Dynamic** subheading that lists each dynamically registered event source. The names of the event sources are the same names as the references you wired to the Register For Events node connected to the Event structure and are listed in the same order. Select the event source you want from the **Dynamic** list. Notice that the same event you registered using the Register For Events node appears highlighted in the **Events** section. After you select the event, edit the case to handle the event data according to the application requirements.

Stop event generation by wiring the dynamic event terminal on the right side of the Event Structure to the **event registration refnum** input of an Unregister For Events function outside the While Loop that contains the Event structure. When the Unregister For Events function executes, LabVIEW unregisters all events the event registration refnum specifies, destroys the event queue associated with the event registration refnum, and discards any events that remain in the queue. If you do not unregister events, and the user performs actions that generate events after the While Loop that contains the Event structure finishes, LabVIEW queues events indefinitely, which can cause the VI to become unresponsive if you configured the events to lock the front panel. In this event, LabVIEW destroys the event queue when the VI goes idle.

LabVIEW also unregisters all events automatically when the top-level VI finishes running. However, National Instruments recommends that you unregister events explicitly, especially in a long-running application, to conserve memory resources.

Dynamic Event Example

The block diagram in Figure 9-1 demonstrates how to use dynamic event registration using the Mouse Enter event on a string control as an example.

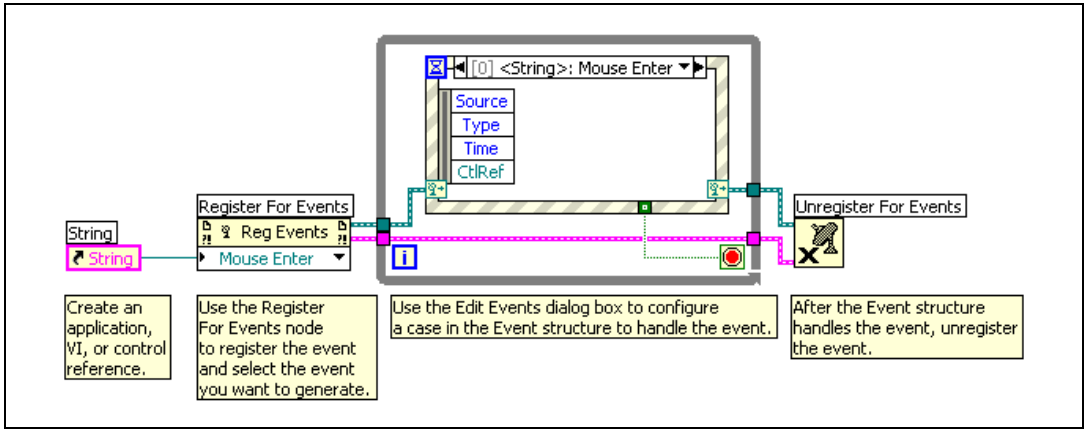


Figure 9-1. Dynamically Registering Events

Refer to the *LabVIEW Help* for more information about how to register events dynamically.

Refer to the `examples\general\dynamicevents.llb` for examples of dynamically registering events.

Modifying Registration Dynamically

If you register for events dynamically, you can modify the registration information at run time to change which objects LabVIEW generates events for. Wire the top left **event registration refnum** input of a Register For Events node if you want to modify the existing registration associated with the refnum you wire rather than create a new registration. When you wire the **event registration refnum** input, the node automatically resizes to show the same events on the same types of references as you specified in the Register For Events node that originally created the event registration refnum. You cannot manually resize or reconfigure the function while the **event registration refnum** input is wired.

If you wire an object reference to an **event source** input of a Register For Events node and you also have wired the **event registration refnum** input, the node replaces whatever reference was previously registered through the corresponding **event source** input of the original Register For Events node. You can wire the Not a Refnum constant to an **event source** input to unregister for an individual event. If you do not wire an **event source** input, LabVIEW does not change the registration for that event. Use the Unregister For Events function if you want to unregister for all events associated with an event registration refnum.

The example in Figure 9-2 shows how you can dynamically change at run time which objects LabVIEW generates events for. When the following block diagram executes, LabVIEW registers the Numeric reference and waits for an event to occur on the associated **Numeric** control. When LabVIEW generates a Value Change event for the **Numeric** control, the Numeric Value Change event case executes a Register For Events node to change the numeric control registered for the Value Change event from **Numeric** to **Numeric 2**. If the user subsequently changes the value of the **Numeric** control, LabVIEW does not generate a Value Change event. However, changes to the **Numeric 2** control generate Value Change events. Each time LabVIEW generates a Value Change event for the **Numeric 2** control, the Register For Events node executes but has no effect because the **Numeric 2** control is already registered for the Value Change event.

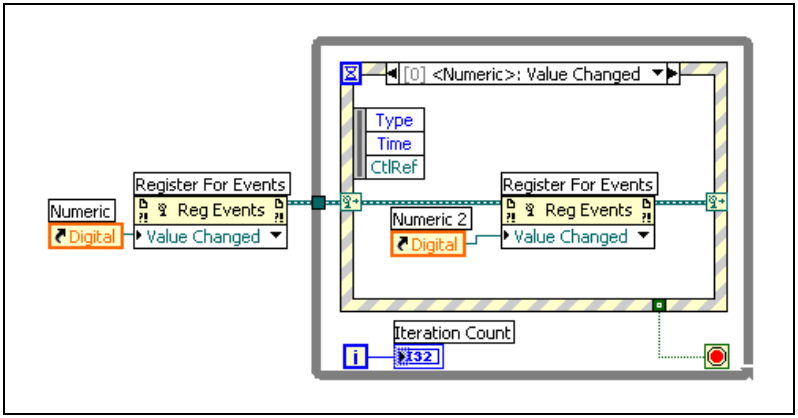


Figure 9-2. Dynamically Modifying Event Registration

Note You cannot dynamically modify statically registered events.

User Events

You programmatically can create and name your own events, called user events, to carry user-defined data. Like queues and notifiers, user events allow different parts of an application to communicate asynchronously. You can handle both user interface and programmatically generated user events in the same Event structure.

Creating and Registering User Events

To define a user event, wire a block diagram object, such as a front panel terminal or block diagram constant, to the Create User Event function. The data type of the object defines the data type of the user event. The label of the object becomes the name of the user event. If the data type is a cluster, the name and type of each field of the cluster define the data the user event carries. If the data type is not a cluster, the user event carries a single value of that type, and the label of the object becomes the name of the user event and of the single data element.

The **user event out** output of the Create User Event function is a strictly typed refnum that carries the name and data type of the user event. Wire the **user event out** output of the Create User Event function to an **event source** input of the Register For Events node.

Handle a user event the same way you handle a dynamically registered user interface event. Wire the **event registration refnum** output of the Register For Events node to the dynamic event terminal on the left side of the Event

structure. Use the **Edit Events** dialog box to configure a case in the Event structure to handle the event. The name of the user event appears under the **Dynamic** subheading in the **Event Sources** section of the dialog box.

The user event data items appear in the Event Data Node on the left border of the Event structure. User events are notify events and can share the same event case of an Event structure as user interface events or other user events.

You can wire a combination of user events and user interface events to the Register For Events node.

Generating User Events

Use the Generate User Event function to deliver the user event and associated data to other parts of an application through an Event structure configured to handle the event. The Generate User Event function accepts a user event refnum and a value for the event data. The data value must match the data type of the user event.

If the user event is not registered, the Generate User Event function has no effect. If the user event is registered but no Event structure is waiting on it, LabVIEW queues the user event and data until an Event structure executes to handle the event. You can register for the same user event multiple times by using separate Register For Event nodes, in which case each queue associated with an event registration refnum receives its own copy of the user event and associated event data each time the Generate User Event function executes.

Note To simulate user interaction with a front panel, you can create a user event that has event data items with the same names and data types as an existing user interface event. For example, you can create a user event called `MyValChg` by using a cluster of two Boolean fields named `OldVal` and `NewVal`, which are the same event data items the Value Change user interface event associates with a Boolean control. You can share the same Event structure case for the simulated `MyValChg` user event and a real Boolean Value Change event. The Event structure executes the event case if a Generate User Event node generates the user event or if a user changes the value of the control.

Unregistering User Events

Unregister user events when you no longer need them. In addition, destroy the user event by wiring the user event refnum to the **user event** input of the Destroy User Event function. Wire the **error out** output of the Unregister For Events function to the **error in** input of the Destroy User Event function to ensure that the functions execute in the correct order.

LabVIEW unregisters all events and destroys existing user events automatically when the top-level VI finishes running. However, National Instruments recommends that you unregister and destroy user events explicitly, especially in a long-running application, to conserve memory resources.

User Event Example

The block diagram in Figure 9-3 shows how to use user events. A block diagram constant cluster specifies the name of the user event, `My Data`, and the data type for the event, a string called `string`. The Register For Events node registers for the user event on the user event refnum. An Event structure in a While Loop waits for the event to occur. In parallel with the While Loop, the Generate User Event function sends the event, which causes the User Event case to execute in the Event structure. When the While Loop terminates, the VI unregisters for the event and destroys the user event.

Build the VI in Figure 9-3 and then use execution highlighting to see how the event data move from node to node through the VI.

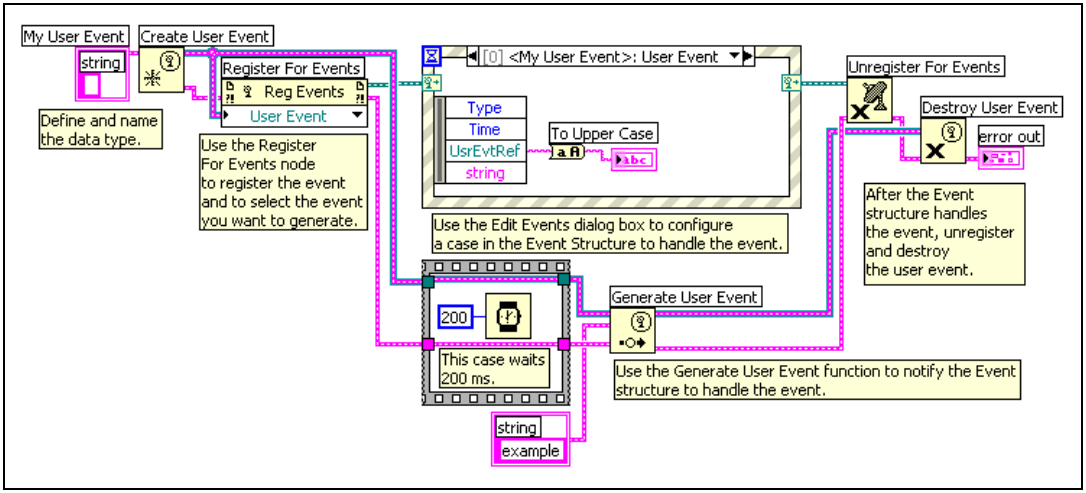


Figure 9-3. Generating User Events

Refer to the `examples\general\dynamicevents.llb` for examples of dynamically registering events.

10

Grouping Data Using Strings, Arrays, and Clusters

Use strings, arrays, and clusters to group data. Strings group sequences of ASCII characters. Arrays group data elements of the same type. Clusters group data elements of mixed types.

For more information…

Refer to the *LabVIEW Help* for more information about grouping data using strings, arrays, and clusters.

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.
- Passing numeric data as character strings to instruments and then converting the strings to numerics.
- Storing numeric data to disk. To store numerics in an ASCII file, you must first convert numerics to strings before writing the numerics 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. Edit and manipulate strings with the String functions on the block diagram. You format strings for use in other applications, such as word processing applications and spreadsheets, or for use within other VIs and functions.

Strings on the Front Panel

Use the string controls and indicators to simulate text entry boxes and labels. Refer to the *String Controls and Indicators* section of Chapter 4, *Building the Front Panel*, for more information about string controls and indicators.

String Display Types

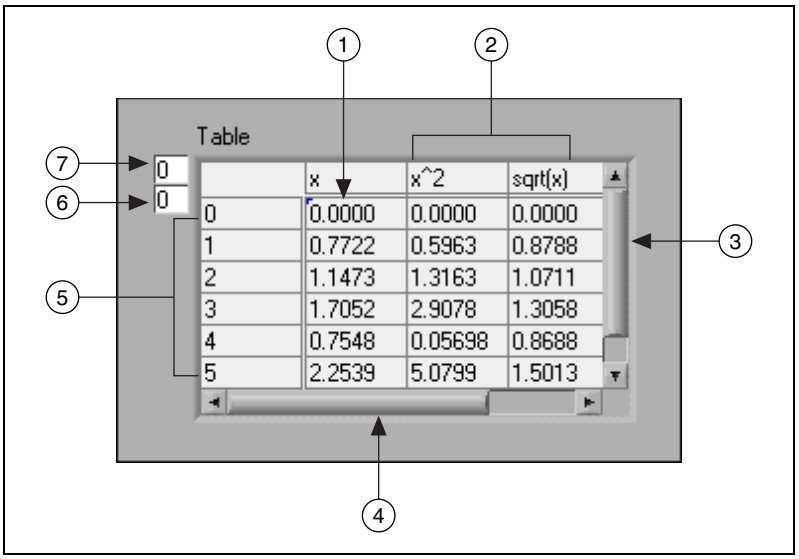
Right-click a string control or indicator on the front panel to select from the display types shown in Table 10-1. The table also shows an example message in each display type.

Table 10-1. 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\st ypes.\n\\\sis\sa\sbackslash.`
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`

Tables

Use the table control to create a table on the front panel. Each cell in a table is a string, and each cell resides in a column and a row. Therefore, a table is a display for a 2D array of strings. Figure 10-1 shows a table and all its parts. Refer to the *Arrays* section of this chapter for more information about arrays.



1	Cell indicated by index values	5	Row headings
2	Column headings	6	Horizontal index
3	Vertical scrollbar	7	Vertical index
4	Horizontal scrollbar		

Figure 10-1. Parts of a Table

Programmatically Editing Strings

Use the String functions to edit strings in the following ways:

- Search for, retrieve, and replace characters or substrings within a string.
- Change all text in a string to upper case or lower case.
- Find and retrieve matching patterns within a string.
- Retrieve a line from a string.
- Rotate and reverse text within a string.
- Concatenate two or more strings.
- Delete characters from a string.

Refer to the `examples\general\strings.llb` for examples of using the String functions to edit strings. Refer to the *Memory and Speed Optimization* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about minimizing memory usage when editing strings programmatically.

Formatting Strings

To use data in another VI, function, or application, you often must convert the data to a string and then format the string in a way that the VI, function, or application can read. For example, Microsoft Excel expects strings that include delimiters, such as tab, commas, or blank spaces. Excel uses the delimiter to segregate numbers or words into cells.

For example, to write a 1D array of numerics to a spreadsheet using the Write File function, you must format the array into a string and separate each numeric with a delimiter, such as a tab. To write an array of numerics to a spreadsheet using the Write to Spreadsheet File VI, you must format the array with the Array to Spreadsheet String function and specify a format and a delimiter.

Use the String functions to perform the following tasks:

- Concatenate two or more strings.
- Extract a subset of strings from a string.
- Convert data into strings.
- Format a string for use in a word processing or spreadsheet application.

Use the File I/O VIs and functions to save strings to text and spreadsheet files.

Format Specifiers

In many cases, you must enter one or more format specifiers in the **format string** parameter of a String function to format a string. A format specifier is a code that indicates how to convert numeric data to or from a string. LabVIEW uses conversion codes to determine the textual format of the parameter. For example, a format specifier of `%x` converts a hex integer to or from a string.

The Format Into String and Scan From String functions can use multiple format specifiers in the **format string** parameter, one for each input or output to the expandable function.

The Array To Spreadsheet String and Spreadsheet String To Array functions use only one format specifier in the **format string** parameter because these functions have only one input to convert. LabVIEW treats any extra specifiers you insert into these functions as literal strings with no special meaning.

Numerics and Strings

Numeric data and string data differ because string data are ASCII characters and numeric data are not. Text and spreadsheet files accept strings only. To write numeric data to a text or spreadsheet file, you must first convert the numeric data to a string.

To add a set of numerics to an existing string, convert the numeric data to a string and use the Concatenate Strings or another String function to add the new string to the existing string. Use the String/Number Conversion functions to convert numerics to strings.

A string can include a set of numerics you display in a graph or chart. For example, you can read a text file that includes a set of numerics that you want to plot to a chart. However, those numerics are in ASCII text, so you must read the numerics as a string and then format the string into a set of numerics before you plot the numerics to a chart.

Figure 10-2 shows a string that includes a set of numerics, converts the string to numerics, builds an array of numerics, and plots the numerics to a graph.

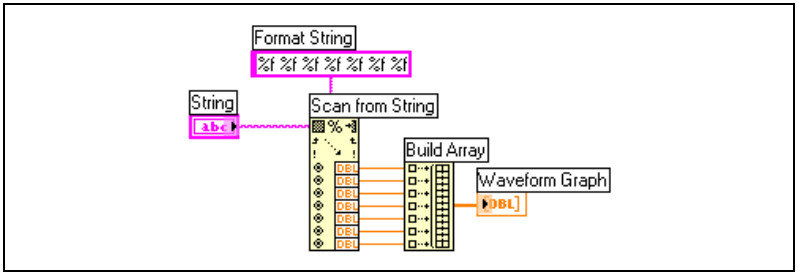


Figure 10-2. Converting a String to Numerics

Converting Data to and from XML

Extensible Markup Language (XML) is a formatting standard that uses tags to describe data. Unlike an HTML tag, an XML tag does not tell a browser how to format a piece of data. Instead, an XML tag identifies a piece of data.

For example, suppose you are a bookseller who sells books on the Web. You want to classify each book in your library by the following criteria:

- Type of book (fiction or nonfiction)
- Title

- Author
- Publisher
- Price
- Genre(s)
- Synopsis
- Number of pages

You can create an XML file for each book. The XML file for a book titled *Touring Germany's Great Cathedrals* would be similar to the following:

```
<nonfiction>
<Title>Touring Germany's Great Cathedrals</Title>
<Author>Tony Walters</Author>
<Publisher>Douglas Drive Publishing</Publisher>
<PriceUS>$29.99</PriceUS>
<Genre>Travel</Genre>
<Genre>Architecture</Genre>
<Genre>History</Genre>
<Synopsis>This book fully illustrates twelve of
Germany's most inspiring cathedrals with full-color
photographs, scaled cross-sections, and time lines of
their construction.</Synopsis>
<Pages>224</Pages>
</nonfiction>
```

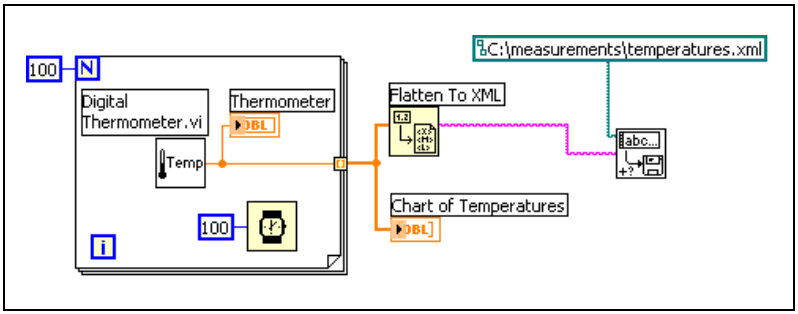
Similarly, you can classify LabVIEW data by name, value, and type. You can represent a string control for a user name in XML as follows:

```
<String>
<Name>User Name</Name>
<Value>Reggie Harmon</Value>
</String>
```

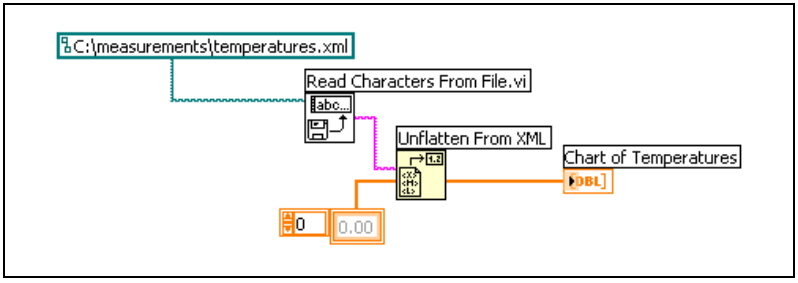
Using XML-Based Data Types

Converting LabVIEW data to XML formats the data so that when you save the data to a file, you easily can identify the value(s), name(s), and type of the data from the tags that describe the data. For example, if you convert an array of temperature values to XML and save that data to a text file, you easily can identify the temperature values by locating the `<Value>` tag that identifies each temperature.

Use the Flatten to XML function to convert a LabVIEW data type to XML format. The following example generates 100 simulated temperatures, plots the array of temperatures to a chart, converts the array of numbers to the XML format, and writes the XML data to the `temperatures.xml` file.



Use the Unflatten from XML function to convert a data type in the XML format into LabVIEW data. The following example reads the 100 temperatures in the `temperatures.xml` file and plots the array of temperatures to a chart.



Note Although you can flatten LabVIEW Variant data to XML, attempting to unflatten variant data from XML results in an empty LabVIEW variant.

Refer to the `examples\file\XMLex.llb` for examples of converting to and from the XML format.

LabVIEW XML Schema

LabVIEW converts data to an established XML schema. Currently, you cannot create customized schemas, and you cannot control how LabVIEW tags each piece of data. Also, you cannot convert entire VIs or functions to XML.

Refer to the `LabVIEW\help` directory for the LabVIEW XML schema.

Grouping Data with Arrays and Clusters

Use the array and cluster controls and functions to group data. Arrays group data elements of the same type. Clusters group data elements of mixed types.

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.

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. Refer to the *Restrictions for Arrays* section of this chapter for more information about the types of elements an array can contain. Refer to the *Clusters* section of this chapter for more information about clusters.

Indexes

To locate a particular element in an array requires one index per dimension. In LabVIEW, indexes let you navigate through an array and retrieve elements, rows, columns, and pages from an array on the block diagram.

Examples of Arrays

An example of a simple array is a text array that lists the nine planets of our solar system. LabVIEW represents this as a 1D array of strings with nine 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, $n = 9$ for the nine planets, so the index ranges from 0 to 8. Earth is the third planet, so it has an index of 2.

Another example of an array is a waveform represented as a numeric array in which each successive element is the voltage value at successive time intervals, as shown in Figure 10-3.

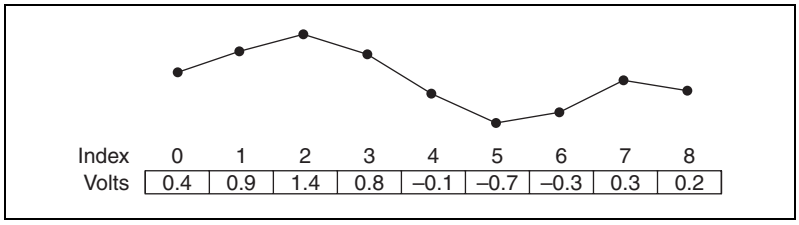


Figure 10-3. Waveform in an Array of Numerics

A more complex example of an array is a graph represented as an array of points where each point is a cluster containing a pair of numerics that represent the X and Y coordinates, as shown in Figure 10-4.

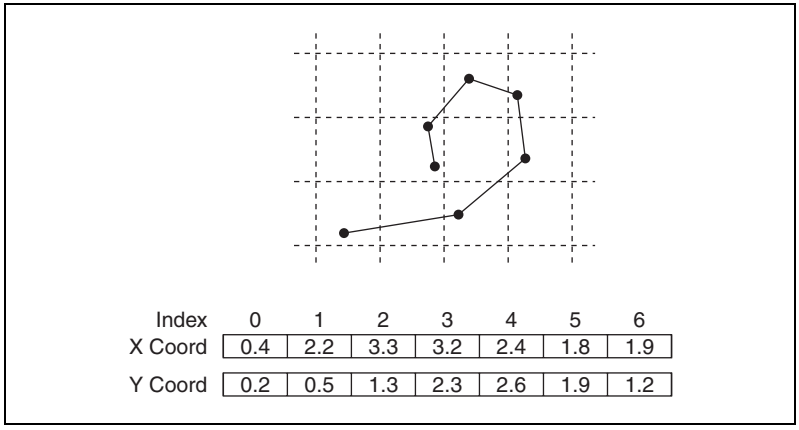


Figure 10-4. Graph in an Array of Points

The previous examples use 1D 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 10-5 shows a 6 column by 4 row 2D array, which contains $6 \times 4 = 24$ elements.

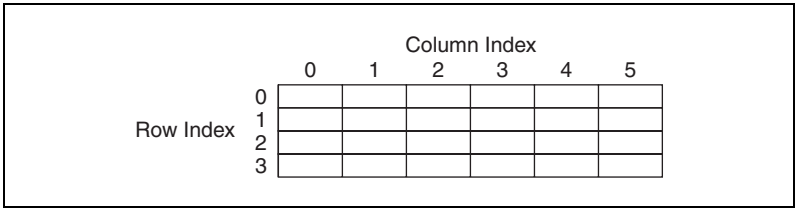
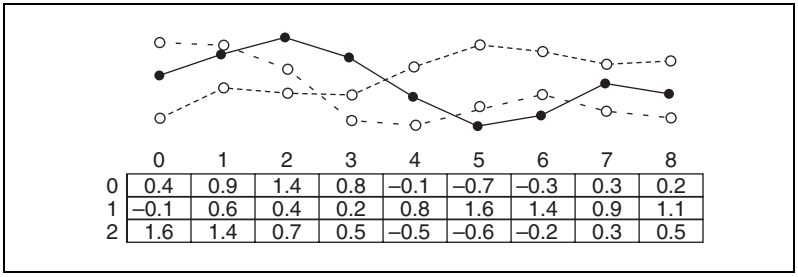


Figure 10-5. 6 Column by 4 Row 2D Array

For example, a chessboard has eight columns and eight rows for a total of 64 positions. Each position can be empty or have one chess piece. You can represent a chessboard as a 2D array of strings. Each string is the name of the piece that occupies the corresponding location on the board or an empty string if the location is empty.

You can generalize the 1D array examples in Figures 10-3 and 10-4 to two dimensions by adding a row to the array. Figure 10-6 shows a collection of waveforms represented as a 2D array of numerics. The row index selects the waveform, and the column index selects the point on the waveform.



	0	1	2	3	4	5	6	7	8
0	0.4	0.9	1.4	0.8	–0.1	–0.7	–0.3	0.3	0.2
1	–0.1	0.6	0.4	0.2	0.8	1.6	1.4	0.9	1.1
2	1.6	1.4	0.7	0.5	–0.5	–0.6	–0.2	0.3	0.5

Figure 10-6. Multiple Waveforms in a 2D Array of Numerics

Refer to the `examples\general\arrays.llb` for examples of using arrays. Refer to the *Using Loops to Build Arrays* section of Chapter 8, *Loops and Structures*, for more information about building arrays.

Restrictions for Arrays

You can create an array of almost any data type, with the following exceptions:

- You cannot create an array of arrays. However, you can use a multidimensional array or use the Build Cluster Array function to create an array of clusters where each cluster contains one or more arrays.
- You cannot create an array of subpanel controls.
- You cannot create an array of tab controls.
- You cannot create an array of ActiveX controls.
- You cannot create an array of charts.
- You cannot create an array of multiplot XY graphs.

Creating Array Controls, Indicators, and Constants

Create an array control or indicator on the front panel by placing an array shell on the front panel, as shown in Figure 10-7, 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.

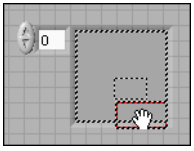


Figure 10-7. Array Shell

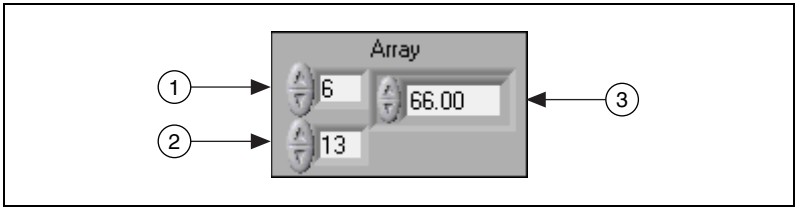
The array shell automatically resizes to accommodate the new object, whether a small Boolean control or a large 3D graph. To create a multidimensional array on the front panel, right-click the index display and select **Add Dimension** from the shortcut menu. You also can resize the index display until you have as many dimensions as you want. To delete dimensions one at a time, right-click the index display and select **Remove Dimension** from the shortcut menu. You also can resize the index display to delete dimensions.

To display a particular element on the front panel, either type the index number in the index display or use the arrows on the index display to navigate to that number.

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

Array Index Display

A 2D array contains rows and columns. As shown in Figure 10-8, the upper display of the two boxes on the left is the row index and the lower display is the column index. The combined display to the right of the row and column displays shows the value at the specified position. Figure 10-8 shows that the value at row 6, column 13, is **66**.



1 Row Index 2 Column Index 3 Value at row, column

Figure 10-8. Array Control

Rows and columns are zero-based, meaning the first column is column 0, the second column is column 1, and so on. Changing the index display for the following array to row 1, column 2 displays a value of 6.

0	1	2	3
4	5	6	7
8	9	10	11

If you try to display a column or row that is out of the range of the array dimensions, the array control appears dimmed to indicate that there is no value defined, and LabVIEW displays the default value of the data type. The default value of the data type depends on the data type of the array.

Use the Positioning tool to resize the array to show more than one row or column at a time.

Array Functions

Use the Array functions to create and manipulate arrays, such as the following tasks:

- Extract individual data elements from an array.
- Insert, delete, or replace data elements in an array.
- Split arrays.

Automatically Resizing Array Functions

The Index Array, Replace Array Subset, Insert Into Array, Delete From Array, and Array Subset functions automatically resize to match the dimensions of the input array you wire. For example, if you wire a 1D array to one of these functions, the function shows a single index input. If you wire a 2D array to the same function, it shows two index inputs—one for the row and one for the column.

You can access more than one element, or subarray (row, column, or page) with these functions by using the Positioning tool to manually resize the function. When you expand one of these functions, the functions expand in increments determined by the dimensions of the array wired to the function. If you wire a 1D array to one of these functions, the function expands by a single index input. If you wire a 2D array to the same function, the function expands by two index inputs—one for the row and one for the column.

The index inputs you wire determine the shape of the subarray you want to access or modify. For example, if the input to an Index Array function is a 2D array and you wire only the **row** input, you extract a complete 1D row of the array. If you wire only the **column** input, you extract a complete 1D column of the array. If you wire the **row** input and the **column** input, you extract a single element of the array. Each input group is independent and can access any portion of any dimension of the array.

The block diagram shown in Figure 10-9 uses the Index Array function to retrieve a row and an element from a 2D array.

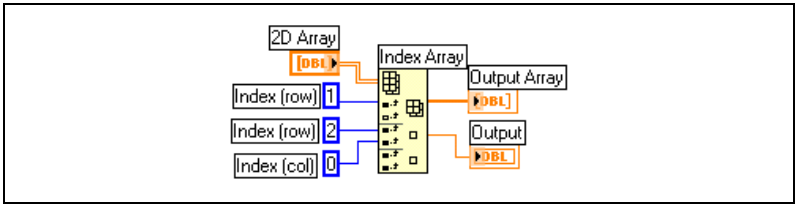


Figure 10-9. Indexing a 2D Array

To access multiple consecutive values in an array, expand the Index Array function, but do not wire values to the index inputs in each increment. For example, to retrieve the first, second, and third rows from a 2D array, expand the Index Array function by three increments and wire 1D array indicators to each **sub-array** output.

Refer to the *Memory and Speed Optimization* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about minimizing memory usage when using Array functions in a loop.

Clusters

Clusters group data elements of mixed types, such as a bundle of wires, as in a telephone cable, where each wire in the cable represents a different element of the cluster. 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 use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Although cluster and array elements are both ordered, you must unbundle all cluster elements at once rather than index one element at a time. You also can use the Unbundle By Name function to access specific cluster elements. 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.

Most clusters on the block diagram have a pink wire pattern and data type terminal. Clusters of numerics, 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.

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.

To wire clusters, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if a double-precision floating-point numeric in one cluster corresponds in cluster order to a string in the another cluster, the wire on the block diagram appears broken and the VI does not run. If numerics of different representations, LabVIEW coerces them to the same representation. Refer to the *Numeric Conversion* section of Appendix B, *Polymorphic Functions*, for more information about numeric conversion.

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

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

11

Local and Global Variables

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 your 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.

For more information…

Refer to the *LabVIEW Help* for more information about using local and global variables.

Local Variables

Use local variables to access front panel objects from more than one location in a single VI and pass data between block diagram structures that you cannot connect with a wire.

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.

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, shown at left, is not yet associated with a control or indicator. Right-click the local variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels.

LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels. Refer to the *Labeling* section of Chapter 4, *Building the Front Panel*, for more information about owned and free labels.

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.

Creating Global Variables

Select a global variable, shown at left, 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. Refer to the *Labeling* section of Chapter 4, *Building the Front Panel*, for more information about owned and free labels.

You can create several single global VIs, each with one front panel object, or you can create one global VI with multiple front panel objects. A global VI with multiple objects is more efficient because you can group related variables together. 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. These global variable nodes are either copies of the first global variable node that you placed on the block diagram of the global VI, or they are the global variable nodes of global VIs that you placed on the current 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 VI associated only with that global variable node and copies of it. Refer to the *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about subVIs.

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. Right-click the global variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels.

Read and Write Variables

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

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

On the block diagram, you can distinguish read locals or globals from write locals or globals the same way you distinguish controls from indicators. A read local or global has a thick border similar to a control. A write local or global has a thin border similar to an indicator.

Refer to the `examples\general\locals.llb` and `examples\general\globals.llb` for examples of using local and global variables.

Using Local and Global Variables Carefully

Local and global variables are advanced LabVIEW concepts. They are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local and global variables, so you should use them carefully. Misusing local and global variables, such as using them instead of a connector pane or using them to access values in each frame of a sequence structure, can lead to unexpected behavior in VIs. Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of data flow, slows performance. Refer to the *Block Diagram Data Flow* section of Chapter 5, *Building the Block Diagram*, for more information about the LabVIEW dataflow execution model.

Initializing Local and Global Variables

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

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

Race Conditions

A race condition occurs when two or more pieces of code that execute in parallel change the value of the same shared resource, typically a local or global variable. Figure 11-1 shows an example of a race condition.

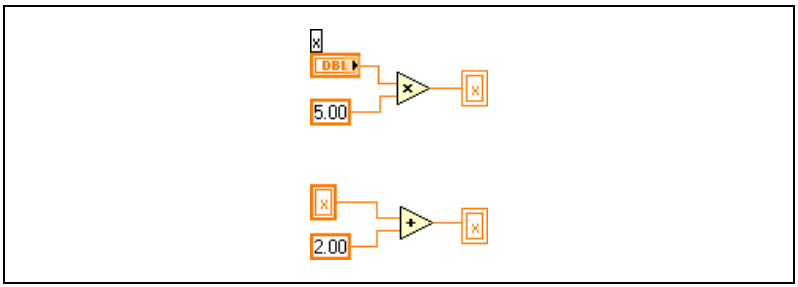


Figure 11-1. Race Condition

The output of this VI depends on the order in which the operations run. Because there is no data dependency between the two operations, there is no way to determine which runs first. To avoid race conditions, do not write to the same variable you read from. Refer to the *Data Dependency and Artificial Data Dependency* section of Chapter 5, *Building the Block Diagram*, for more information about data dependency.

Memory Considerations when Using Local Variables

Local variables make copies of data buffers. When you read from a local variable, you create a new buffer for the data from its associated control.

If you use local variables to transfer large amounts of data from one place on the block diagram to another, you generally use more memory and, consequently, have slower execution speed than if you transfer data using a wire. If you need to store data during execution, consider using a shift register.

Memory Considerations when Using Global Variables

When you read from a global variable, LabVIEW creates a copy of the data stored in that global variable.

When you manipulate large arrays and strings, the time and memory required to manipulate global variables can be considerable. Manipulating global variables is especially inefficient when dealing with arrays because if you modify only a single array element, LabVIEW stores and modifies the entire array. If you read from the global variable in several places in an application, you create several memory buffers, which is inefficient and slows performance.

12

Graphs and Charts

Use graphs and charts to display data in a graphical form.

Graphs and charts differ in the way they display and update data. VIs with graphs usually collect the data in an array and then plot the data to the graph, which is similar to a spreadsheet that first stores the data then generates a plot of it. In contrast, a chart appends new data points to those already in the display to create a history. On a chart, you can see the current reading or measurement in context with data previously acquired.

For more information…

Refer to the *LabVIEW Help* for more information about using graphs and charts.

Types of Graphs and Charts

The graphs and charts include the following types:

- **Waveform Chart and Graph**—Displays data acquired at a constant rate.
- **XY Graph**—Displays data acquired at a non-constant rate, such as data acquired when a trigger occurs.
- **Intensity Chart and Graph**—Displays 3D data on a 2D plot by using color to display the values of the third dimension.
- **Digital Waveform Graph**—Displays data as pulses or groups of digital lines. Computers transfer digital data to other computers in pulses.
- **(Windows) 3D Graphs**—Displays 3D data on a 3D plot in an ActiveX object on the front panel.

Refer to `examples\general\graphs` for examples of graphs and charts.

Graph and Chart Options

Although graphs and charts plot data differently, they have several common options that you access from the shortcut menu. Refer to the *Customizing Graphs* and *Customizing Charts* sections of this chapter for more information about the options available only on graphs or only on charts.

Waveform and XY graphs and charts have different options than intensity, digital, and 3D graphs and charts. Refer to the *Intensity Graphs and Charts*, *3D Graphs*, and *Digital Waveform Graphs* sections of this chapter for more information about intensity, digital, and 3D graph and chart options.

Multiple X- and Y-Scales on Graphs and Charts

All graphs support multiple x- and y-scales, and all charts support multiple y-scales. Use multiple scales on a graph or chart to display multiple plots that do not share a common x- or y-scale. Right-click the scale of the graph or chart and select **Duplicate Scale** from the shortcut menu to add multiple scales to the graph or chart.

Anti-Aliased Line Plots for Graphs and Charts

You can improve the appearance of line plots in graphs and charts by using anti-aliased lines. When you enable anti-aliased line drawing, line plots appear smoother. Anti-aliased line drawing does not alter line widths, line styles, point styles, and so on.

Note Anti-aliased line drawing is not available on digital waveform graphs.

To enable anti-aliased line plots, right-click the plot legend and select **Anti-aliased** from the shortcut menu. If the plot legend is not visible, right-click the graph or chart and select **Visible Items»Plot Legend** from the shortcut menu.

Note Because anti-aliased line drawing can be computation intensive, using anti-aliased line plots can slow performance.

Customizing Graph and Chart Appearance

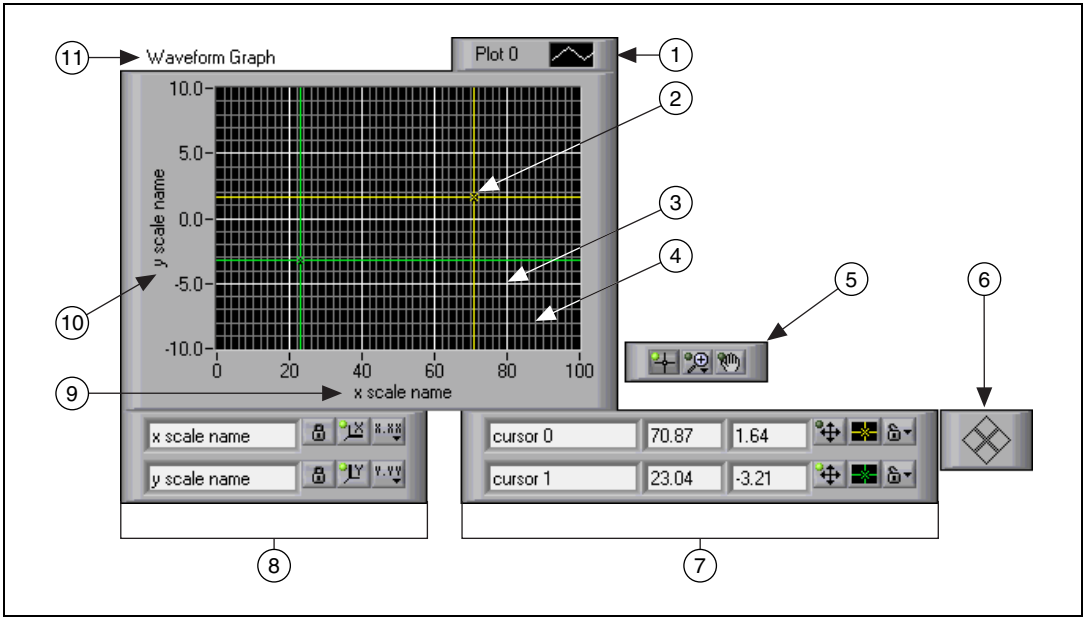
You customize the appearance of graphs and charts by showing or hiding options. Right-click the graph or chart and select **Visible Items** from the shortcut menu to display or hide the following options:

- **Plot Legend**—Defines the color and style of the plot(s). Resize the legend to display multiple plots.
- **Scale Legend**—Defines labels for scales and configures scale properties.
- **Graph Palette**—Changes scaling and formatting while a VI is running.
- **X Scale and Y Scale**—Formats the x- and y-scales. Refer to the *Axis Formatting* section of this chapter for more information about scales.
- **Cursor Legend (graph only)**—Displays a marker at a defined point coordinate. You can display multiple cursors on a graph.
- **X Scrollbar**—Scrolls through the data in the graph or chart. Use the scrollbar to view data that the graph or chart does not currently display.

Customizing Graphs

You can modify the behavior of graph cursors, scaling options, and graph axes. Figure 12-1 illustrates the elements of a graph.

You can customize the waveform and intensity graphs to match your data display requirements or to display more information. Options available for charts include a scrollbar, a legend, a palette, a digital display, and representation of scales with respect to time.



1	Plot legend	4	Mini-grid mark	7	Cursor legend	10	Y-scale
2	Cursor (graph only)	5	Graph palette	8	Scale legend	11	Label
3	Grid mark	6	Cursor mover	9	X-scale		

Figure 12-1. Graph Elements

You add most of the items listed in the legend above by right-clicking the graph, selecting **Visible Items** from the shortcut menu, and selecting the appropriate element.

Graph Cursors

Cursors on graphs allow you to read the exact value of a point on a plot. The cursor value displays in the cursor legend. Add a cursor to a graph by right-clicking the graph, selecting **Visible Items»Cursor Legend** from the shortcut menu, and clicking anywhere in a cursor legend row to activate a cursor. Use the Positioning tool to expand the cursor legend to add multiple cursors.

You can place cursors and a cursor display on all graphs, and you can label the cursor on the plot. You can set a cursor to lock onto a plot, and you can move multiple cursors at the same time. A graph can have any number of cursors.

Autoscaling

Graphs can automatically adjust their horizontal and vertical scales to reflect the data you wire to them. This behavior is called autoscaling. Turn autoscaling on or off by right-clicking the graph and selecting **X Scale» Autoscale X** or **Y Scale»Autoscale Y** from the shortcut menu. By default, autoscaling is enabled for graphs. However, autoscaling can slow performance.

Use the Operating or Labeling tool to change the horizontal or vertical scale directly.

Waveform Graph Scale Legend

Use the scale legend to label scales and configure scale properties.

Use the Operating tool to click the **Scale Format** button, shown at left, to configure the format, precision, and mapping mode.

Use the **Scale Lock** button, shown at left, to toggle autoscaling for each scale, the visibility of scales, scale labels, and plots, and to format scale labels, grids, grid lines, and grid colors.

Axis Formatting

Use the **Format and Precision** tab of the **Graph Properties** and **Chart Properties** dialog boxes to specify how the x-axis and y-axis appear on the graph or chart.

By default, the x-axis is configured to use floating-point notation and with a label of `Time`, and the y-axis is configured to use automatic formatting and with a label of `Amplitude`. Right-click the graph or chart and select **Properties** to display the **Graph Properties** dialog box or **Chart Properties** dialog box to configure the axes for the graph or chart.

Use the **Format and Precision** tab of the **Graph Properties** or **Chart Properties** dialog box to specify a numeric format for the axes of a graph or chart. Select the **Scales** tab to rename the axis and to format the appearance of the axis scale. By default, a graph or chart axis displays up to six digits before automatically switching to exponential notation.

Place a checkmark in the **Direct entry mode preferred** checkbox to display the text options that let you enter format strings directly. Enter format strings to customize the appearance of the scales and numeric precision of the axes. Refer to the *Using Format Strings* topic of the *LabVIEW Help* for more information about formatting strings.

Dynamically Formatting Graphs

Wire a dynamic data type output to a waveform graph to automatically format the plot legend and x-scale time stamp for the graph. For example, if you configure the Simulate Signal Express VI to generate a sine wave and to use absolute time and wire the output of the Simulate Signal Express VI to a waveform graph, the plot legend of the graph automatically updates the plot label to `Sine`, and the x-scale displays the time and date when you run the VI.

Right-click the graph and select **Ignore Attributes** from the shortcut menu to ignore the plot legend label the dynamic data type includes. Right-click the graph and select **Ignore Time Stamp** from the shortcut menu to ignore the time stamp configuration the dynamic data type includes.

Note If the data you display in the waveform graph contains an absolute time time stamp, select **Ignore Time Stamp** to ignore the time stamp. Refer to the *Getting Started with LabVIEW* manual for more information about using the dynamic data type with graphs and for more information about Express VIs. Refer to the *Dynamic Data Type* section of Chapter 5, *Building the Block Diagram*, for more information about using the dynamic data type.

Using Smooth Updates

When LabVIEW updates an object with smooth updates off, it erases the contents of the object and draws the new value, which results in a noticeable flicker. Using smooth updates avoids the flicker that erasing and drawing causes. Right-click the graph and select **Advanced»Smooth Updates** from the shortcut menu to use an offscreen buffer to minimize the flicker. Using **Smooth Updates** can slow performance depending on the computer and video system you use.

Customizing Charts

Unlike graphs, which display an entire waveform that overwrites the data already stored, charts update periodically and maintain a history of the data previously stored.

You can customize the waveform and intensity charts to match your data display requirements or to display more information. Options available for charts include a scrollbar, a legend, a palette, a digital display, and representation of scales with respect to time. You can modify the behavior of chart history length, update modes, and plot displays.

Chart History Length

LabVIEW stores data points already added to the chart in a buffer, or the chart history. The default size for a chart history buffer is 1,024 data points. You can configure the history buffer by right-clicking the chart and selecting **Chart History Length** from the shortcut menu. You can view previously collected data using the chart scrollbar. Right-click the chart and select **Visible Items»X Scrollbar** from the shortcut menu to display a scrollbar.

Chart Update Modes

Charts use the following three different modes to display data. Right-click the chart and select **Advanced»Update Mode** from the shortcut menu. Select **Strip Chart**, **Scope Chart**, or **Sweep Chart**. The default mode is **Strip Chart**.

- **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.
- **Scope Chart**—Shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to the 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.

Overlaid versus Stacked Plots

You can display multiple plots on a chart by using a single vertical scale, called overlaid plots, or by using multiple vertical scales, called stacked plots. Figure 12-2 shows examples of overlaid plots and stacked plots.

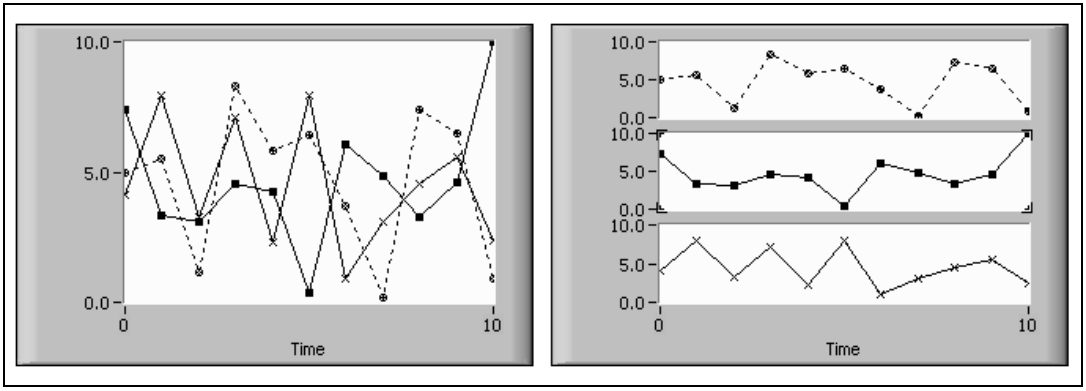


Figure 12-2. Charts with Overlaid and Stacked Plots

Right-click the chart and select **Stack Plots** from the shortcut menu to view the chart plots as multiple vertical scales. Right-click the chart and select **Overlay Plots** to view the chart plots as a single vertical scale.

Refer to the Charts VI in the `examples\general\graphs\charts.llb` for examples of different kinds of charts and the data types they accept.

Waveform and XY Graphs

Waveform graphs display evenly sampled measurements. XY graphs display any set of points, evenly sampled or not. Figure 12-3 shows examples of a waveform graph and an XY graph.

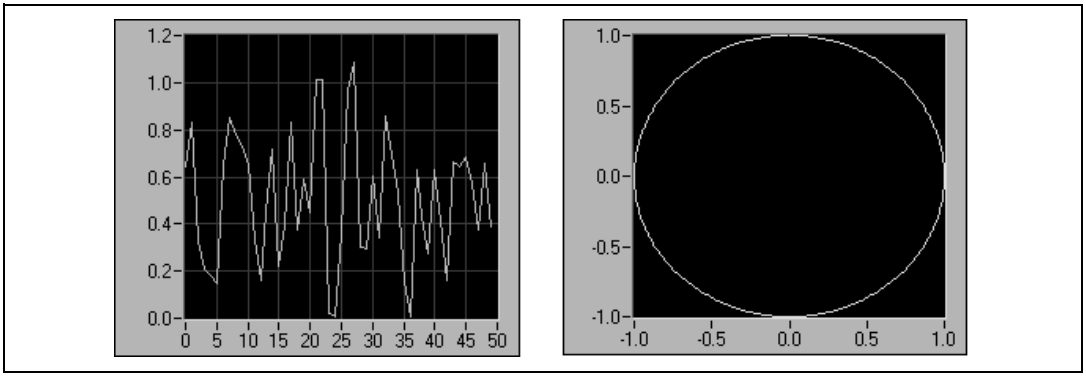


Figure 12-3. Waveform and XY Graphs

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. The XY graph is a general-purpose, Cartesian graphing object that plots multivalued functions, such as circular shapes or waveforms with a varying timebase. Both graphs can display plots containing any number of points.

Both types of graphs accept several data types, which minimizes the extent to which you must manipulate data before you display it.

Single-Plot Waveform Graph Data Types

The waveform graph accepts two data types for single-plot waveform graphs.

The graph accepts a single array of values and interprets the data as points on the graph and increments the x index by one starting at $x = 0$. The graph also accepts a cluster of an initial x value, a Δx, and an array of y data.

Refer to the Waveform Graph VI in the `examples\general\graphs\gengraph.llb` for examples of the data types that single-plot waveform graphs accept.

Multiplot Waveform Graph

A multiplot 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 returns 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 `examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

A multiplot waveform graph also accepts a cluster of an x value, a Δx 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 Δx, starting at $x = 0$. This data type is useful for displaying multiple signals all sampled at the same regular rate. Refer to the (Xo, dX, Y) Multi Plot 3 graph in the Waveform Graph VI in the `examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

A multiplot waveform graph accepts a plot array where the array contains clusters. Each cluster contains a point 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. Figure 12-4 shows this array of the **y** cluster.

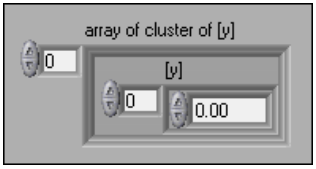


Figure 12-4. Array of Cluster y

Use a multiplot waveform graph 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 `examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

A multiplot waveform graph accepts a cluster of an initial *x* value, a Δx value, and array that contains clusters. Each cluster contains a point 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. You also can use the Build Cluster Array, which creates arrays of clusters that contain inputs you specify. Refer to the (Xo, dX, Y) Multi Plot 2 graph in the Waveform Graph VI in the `examples\general\graphs\gengraph.llb` for an example of a graph that accepts this data type.

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

Single-Plot XY Graph Data Types

The single-plot XY graph accepts a cluster that contains an *x* array and a *y* array. 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 XY Graph VI in the `examples\general\graph\gengraph.llb` for an example of single-plot XY graph data types.

Multiplot XY Graph Data Types

The multiplot XY graph accepts an array of plots, where a plot is a cluster that contains an *x* array and a *y* array. The multiplot 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 XY Graph VI in the `examples\general\graph\gengraph.llb` for an example of multiplot XY graph data types.

Waveform Charts

The waveform chart is a special type of numeric indicator that displays one or more plots. Refer to the `examples\general\graphs\charts.llb` for examples of waveform charts.

If you pass charts a single value or multiple values at a time, LabVIEW interprets the data as points on the chart and increments the *x* index by one starting at $x = 0$. The chart treats these inputs as new data for a single plot. If you pass charts the waveform data type, the *x* index conforms to the time format specified.

The frequency at which you send data to the chart determines how often the chart redraws.

To pass data for multiple plots to a waveform chart, you can bundle the data together into a cluster of scalar numerics, where each numeric represents a single point for each of the plots.

If you want to pass multiple points for plots in a single update, wire an array of clusters of numerics to the chart. Each numeric represents a single *y* value point for each of the plots.

If you cannot determine the number of plots you want to display until run time, or you want to pass multiple points for multiple plots in a single update, wire a 2D array of numerics or waveforms to the chart. As with the waveform graph, by default, waveform charts handle rows as new data for each plot. Wire a 2D array data type to the chart, right-click the chart, and select **Transpose Array** from the shortcut menu to treat columns in the array as new data for each plot.

Intensity Graphs and Charts

Use the intensity graph and chart to display 3D data on a 2D plot by placing blocks of color on a Cartesian plane. For example, you can use intensity graphs and charts to display patterned data, such as temperature patterns and terrain, where the magnitude represents altitude. The intensity graph and chart accept a 2D array of numbers. Each number in the array represents a specific color. The indexes of the elements in the 2D array set the plot locations for the colors. Figure 12-5 shows the concept of the intensity chart operation.

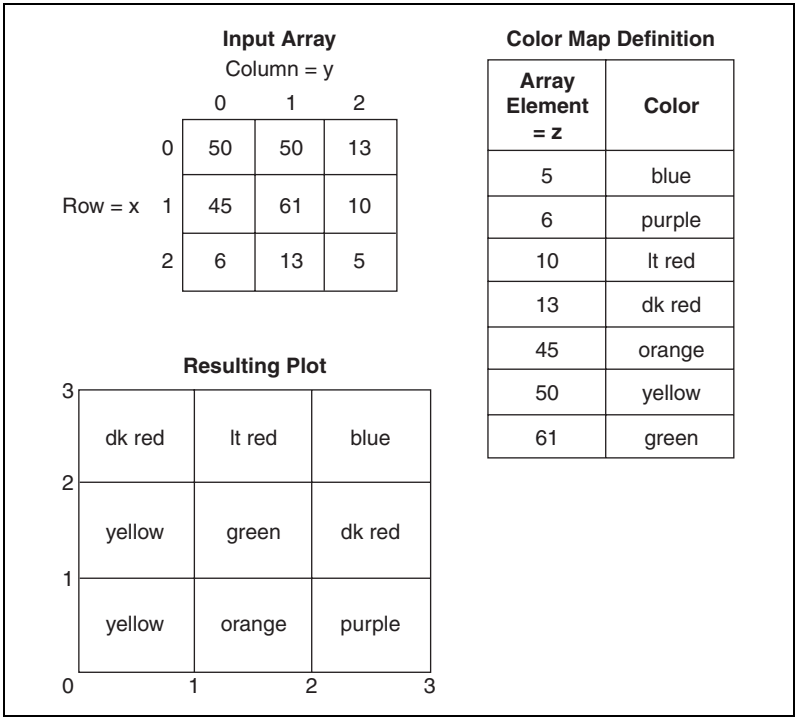


Figure 12-5. Intensity Chart Color Map

The rows of the data pass into the display as new columns on the graph or chart. If you want rows to appear as rows on the display, wire a 2D array data type to the graph or chart, right-click the graph or chart, and select **Transpose Array** from the shortcut menu.

The array indexes correspond to the lower left vertex of the block of color. The block of color has a unit area, which is the area between the two points,

as defined by the array indexes. The intensity graph or chart can display up to 256 discrete colors.

After you plot a block of data on an intensity chart, the origin of the Cartesian plane shifts to the right of the last data block. When the chart processes new data, the new data appear to the right of the old data. When a chart display is full, the oldest data scroll off the left side of the chart. This behavior is similar to the behavior of strip charts. Refer to the *Chart Update Modes* section of this chapter for more information about these charts.

Refer to the `examples\general\graphs\intgraph.llb` for examples of intensity charts and graphs.

Color Mapping

You can set the color mapping interactively for intensity graphs and charts the same way you define the colors for a color ramp numeric control. Refer to the *Color Ramps* section of Chapter 4, *Building the Front Panel*, for more information about color ramps.

You can set the color mapping for the intensity graph and chart programmatically by using the Property Node in two ways. Typically, you specify the value-to-color mappings in the Property Node. For this method, specify the Z Scale: Marker Values property. This property consists of an array of clusters, in which each cluster contains a numeric limit value and the corresponding color to display for that value. When you specify the color mapping in this manner, you can specify an upper out-of-range color using the Z Scale: High Color property and a lower out-of-range color using the Z Scale: Low Color property. The intensity graph and chart is limited to a total of 254 colors, with the lower and upper out-of-range colors bringing the total to 256 colors. If you specify more than 254 colors, the intensity graph or chart creates the 254-color table by interpolating among the specified colors.

If you display a bitmap on the intensity graph, you specify a color table using the Color Table property. With this method, you can specify an array of up to 256 colors. Data passed to the chart are mapped to indexes in this color table based on the color scale of the intensity chart. If the color scale ranges from 0 to 100, a value of 0 in the data is mapped to index 1, and a value of 100 is mapped to index 254, with interior values interpolated between 1 and 254. Anything below 0 is mapped to the out-of-range below color (index 0), and anything above 100 is mapped to the out-of-range above color (index 255).

Note The colors you want the intensity graph or chart to display are limited to the exact colors and number of colors your video card can display. You also are limited by the number of colors allocated for your display.

Intensity Chart Options

The intensity chart shares many of the optional parts of the waveform charts, which you can show or hide by right-clicking the chart and selecting **Visible Items** from the shortcut menu. In addition, because the intensity chart includes color as a third dimension, a scale similar to a color ramp control defines the range and mappings of values to colors.

Like waveform charts, the intensity chart maintains a history of data, or buffer, from previous updates. You can configure this buffer by right-clicking the chart and selecting **Chart History Length** from the shortcut menu. The default size for an intensity chart is 128 data points. The intensity chart display can be memory intensive. For example, displaying a single-precision chart with a history of 512 points and 128 *y* values requires 512 * 128 * 4 bytes (size of a single-precision number), or 256 KB.

Intensity Graph Options

The intensity graph works the same as the intensity chart, except it does not retain previous data and does not include update modes. Each time new data pass to an intensity graph, the new data replace old data.

The intensity graph can have cursors like other graphs. Each cursor displays the *x*, *y*, and *z* values for a specified point on the graph.

Digital Waveform Graphs

Use the digital waveform graph to display digital data, especially when you work with timing diagrams or logic analyzers. Refer to the *LabVIEW Measurements Manual* for more information about acquiring digital data.

The digital waveform graph accepts the digital waveform data type, the digital data type, and an array of those data types as an input. By default, the digital waveform graph collapses digital buses, so the graph plots digital data on a single plot. If you wire an array of digital data, the digital waveform graph plots each element of the array as a different plot in the order of the array.

The digital waveform graph in Figure 12-6 plots digital data on a single plot. The VI converts the numbers in the **Numbers** array to digital data and displays the binary representations of the numbers in the **Binary Representations** digital data indicator. In the digital graph, the number 0 appears without a top line to symbolize that all the bit values are zero. The number 255 appears without a bottom line to symbolize that all the bit values are 1.

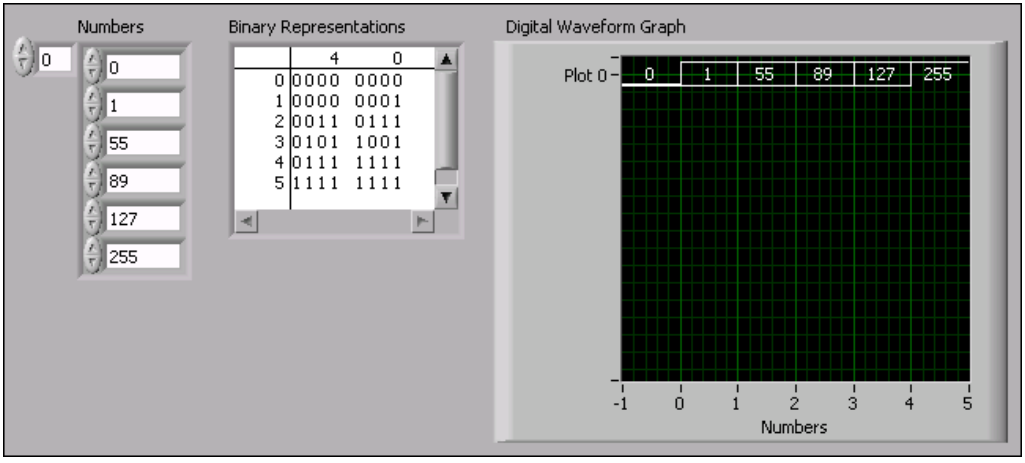


Figure 12-6. Plotting Digital Data on a Single Plot

Refer to the *LabVIEW Help* for more information about configuring the plots of the digital waveform graph.

To plot each sample of digital data, right-click the y-axis and select **Expand Digital Buses** from the shortcut menu. Each plot represents a different sample.

The digital waveform graph in Figure 12-7 displays the six numbers in the **Numbers** digital array control.

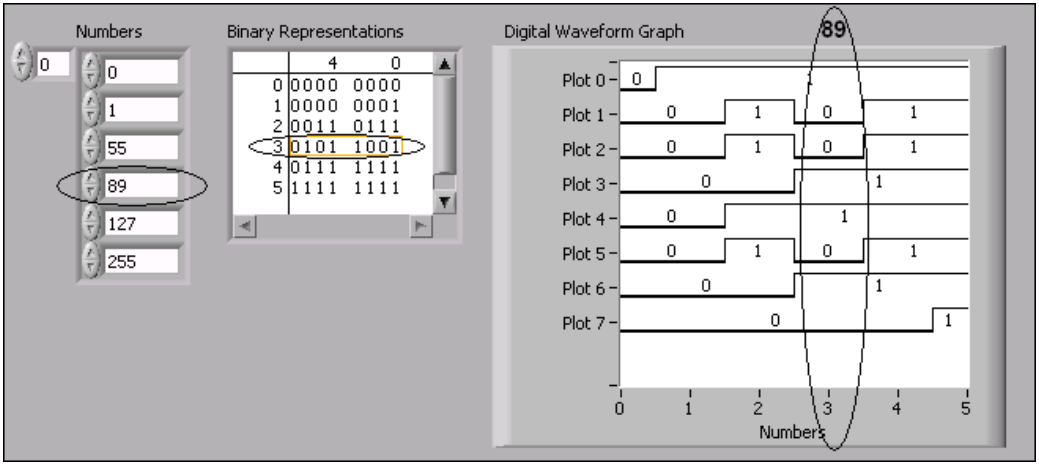


Figure 12-7. Graphing Integers Digitally

The **Binary Representations** digital control displays the binary representations of the numbers. Each column in the table represents a bit. For example, the number 89 requires 7 bits of memory (the 0 in column 7 indicates an unused bit). Point 3 on the digital waveform graph plots the 7 bits necessary to represent the number 89 and a value of zero to represent the unused eighth bit on plot 7.

The VI in Figure 12-8 converts an array of numbers to digital data and uses the Build Waveform function to assemble the start time, delta t, and the numbers entered in a digital data control and to display the digital data.

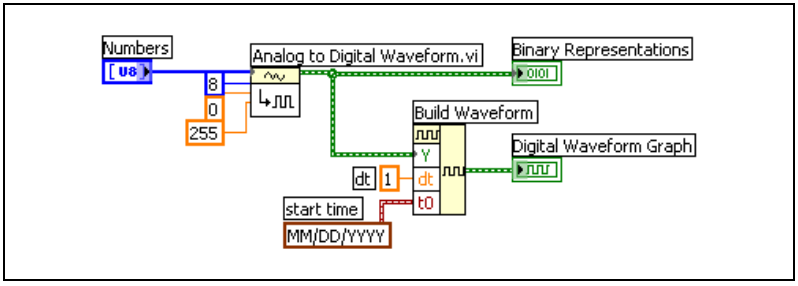


Figure 12-8. Converting an Array of Numbers to Digital Data

Refer to the *Digital Waveform Control* and *Digital Data Control* sections in Chapter 4, *Building the Front Panel*, for more information about the digital data control, the digital waveform data type, and converting data to digital data.

Masking Data

The VI in Figure 12-7 produces a graph in which each plot represents one bit in the data. You also can select, reorder, and combine bits in the data before you display it on the graph. Selecting, reordering, and combining bits is called masking the data.

Use a mask to combine the plots of two or more different bits and display them on a single plot. If you have an array of 8-bit integers, you can plot up to 8 bits on a single plot. If you have an array of 16-bit integers, you can display up to 16 bits on a single plot, and so on. You also can plot the same bit two or more times on a single plot.

3D Graphs

Note 3D graph controls are available only on Windows in the LabVIEW Full and Professional Development Systems.

For many real-world data sets, such as the temperature distribution on a surface, joint time-frequency analysis, and the motion of an airplane, you need to visualize data in three dimensions. With the 3D graphs, you can visualize three-dimensional data and alter the way that data appears by modifying the 3D graph properties.

The following 3D graphs are available:

- **3D Surface**—Draws a surface in 3D space. When you drop this control on the front panel, LabVIEW wires it to a subVI that receives the data that represent the surface.
- **3D Parametric**—Draws a complex surface in 3D space. When you drop this control on the front panel, LabVIEW wires it to a subVI that receives the data that represent the surface.
- **3D Curve**—Draws a line in 3D space. When you drop this control on the front panel, LabVIEW wires it to a subVI that receives the data that represent the line.

Use the 3D graphs in conjunction with the 3D Graph VIs to plot curves and surfaces. A curve contains individual points on the graph, each point having an *x*, *y*, and *z* coordinate. The VI then connects these points with a line. A curve is ideal for visualizing the path of a moving object, such as the flight path of an airplane.

The 3D graphs use ActiveX technology and VIs that handle 3D representation. You can set properties for the 3D Graph Properties VIs to change behavior at run time, including setting basic, axes, grid, and projection properties.

A surface plot uses *x*, *y*, and *z* data to plot points on the graph. The surface plot then connects these points, forming a three-dimensional surface view of the data. For example, you could use a surface plot for terrain mapping.

When you select a 3D graph, LabVIEW places an ActiveX container on the front panel that contains a 3D graph control. LabVIEW also places a reference to the 3D graph control on the block diagram. LabVIEW wires this reference to one of the three 3D Graph VIs.

Waveform Data Type

The waveform data type carries the data, start time, and Δt of a waveform. You can create waveforms using the Build Waveform function. Many of the VIs and functions you use to acquire or analyze waveforms accept and return the waveform data type by default. When you wire a waveform data type to a waveform graph or chart, the graph or chart automatically plots a waveform based on the data, start time, and Δx of the waveform. When you wire an array of waveform data types to a waveform graph or chart, the graph or chart automatically plots all the waveforms. Refer to Chapter 5, *Creating a Typical Measurement Application*, of the *LabVIEW Measurements Manual* for more information about using the waveform data type.

Digital Waveform Data Type

The digital waveform data type carries start time, delta x, the data, and the attributes of a digital waveform. You can use the Build Waveform function to create digital waveforms. When you wire a digital waveform data type to the digital waveform graph, the graph automatically plots a waveform based on the timing information and data of the digital waveform. Wire the digital waveform data type to a digital data indicator to view the samples and signals of a digital waveform. Refer to Chapter 5, *Typical Measurement Applications*, of the *LabVIEW Measurements Manual* for more information about using the digital waveform data type.

13

Graphics and Sound VIs

Use the Graphics and Sound VIs to display or modify graphics and sound in VIs.

For more information…

Refer to the *LabVIEW Help* for more information about using graphics and sound in VIs.

Using the Picture Indicator

The picture indicator is a general-purpose indicator for displaying pictures that can contain lines, circles, text, and other types of graphic shapes. Because you have pixel-level control over the picture indicator, you can create nearly any graphics object. Use the picture indicator and the graphics VIs instead of a graphics application to create, modify, and view graphics in LabVIEW.

The picture indicator has a pixel-based coordinate system in which the origin (0, 0) is the pixel located at the top left corner of the control. The horizontal (*x*) component of a coordinate increases toward the right, and the vertical (*y*) coordinate increases toward the bottom.

If a picture is too large for the picture indicator that displays it, LabVIEW crops the picture so you can see only the pixels that fit within the display region of the indicator. Use the Positioning tool to resize the indicator and run the VI again to view the entire picture. You also can display vertical and horizontal scrollbars for a picture indicator so you can view the pixels that do not fit within the display region of the indicator. Right-click the indicator and select **Visible Items»Scrollbar** from the shortcut menu to display scrollbars for the indicator.

You can use VI Server properties in the Picture class to modify the picture indicator programmatically, such as to change the size of the picture indicator or the picture area in the indicator. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using VI Server properties.

When you place a picture indicator on the front panel, it appears as a blank rectangular area, and a corresponding terminal, shown at left, appears on the block diagram.

To display an image in a picture indicator, you must write an image to the indicator programmatically. You cannot copy an image to the clipboard and paste it into the picture indicator. You can use the Picture Functions VIs to specify a set of drawing instructions. Each VI takes a set of inputs that describes a drawing instruction. Based on these inputs, the VI creates a compact description of these specifications that you pass to the picture indicator for display. Refer to the *Picture Functions VIs* section of this chapter for more information about using the Picture Functions VIs to display an image in a picture indicator.

Picture Plots VIs

Use the Picture Plot VIs to create common types of graphs using the picture indicator. These graphs include a polar plot, a waveform graph, an XY graph, a Smith plot, a radar plot, and a graph scale.

The Picture Plot VIs use low-level drawing functions to create a graphical display of your data and customize the drawing code to add functionality. These graphical displays are not as interactive as the built-in LabVIEW controls, but you can use them to visualize information in ways the built-in controls currently cannot. For example, you can use the Plot Waveform VI to create a plot with slightly different functionality than built-in waveform graphs.

Using the Polar Plot VI as a SubVI

Use the Polar Plot VI to draw specific, contiguous quadrants of a polar graph or the entire graph at once. As with the built-in LabVIEW graphs, you can specify the color of the components, include a grid, and specify the range and format for the scales.

The Polar Plot VI provides a large amount of functionality in a single VI. Consequently, the VI includes complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VI. Instead of creating the default cluster input on the block diagram, copy a custom control from the Polar Plot Demo VI in the `examples\picture\demos.llb` and place it on the front panel.

Using the Plot Waveform and Plot XY VIs as SubVIs

Use the Plot Waveform VI, which emulates the behavior of the built-in waveform graph, to draw waveforms in a variety of styles, including points, connected points, and bars. As with the built-in waveform graphs, you can specify the color of the components, include a grid, and specify the range and format for the scales.

The Plot Waveform VI provides a large amount of functionality in a single VI. Consequently, the VI includes complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VI. Instead of creating the default cluster input, copy a custom control from the Waveform and XY Plots VI in the `examples\picture\demos.llb` and place it on the front panel.

The Plot XY VI and the Plot Multi-XY VI are similar to the Plot Waveform VI. You use different controls to specify the cosmetic appearance of the plot because the XY plotting VIs have three additional plot styles—two scatter plot styles and a plot style where a line is drawn at each unique *x*-position to mark the minimum and maximum *y*-values for that *x*-value.

Using the Smith Plot VIs as SubVIs

Use the Smith Plot VIs to study transmission line behavior, such as in the telecommunications industry. A transmission line is a medium through which you transmit energy and signals. A transmission line can be a wire or it can be the atmosphere through which a signal transmits. Transmission lines have an effect on the signal that is transmitting. This effect, called the impedance of the transmission line, can attenuate or phase shift an AC signal.

The impedance of the transmission line is a measure of the resistance and the reactance of the line. The impedance, z, is commonly listed as a complex number of the form $z = r + jx$, where both resistance (r) and reactance (x) are components.

Use the Smith Plot VIs to display impedances of transmission lines. The plot consists of circles of constant resistance and reactance.

You can plot a given impedance, $r + jx$, by locating the intersection of the appropriate r circle and x circle. After you plot the impedance, use the Smith Plot VIs as visual aids to match impedance and to calculate the reflection coefficient of a transmission line.

The Smith Plot VIs provide a large amount of functionality in each single VI. Consequently, many of these VIs include complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VIs. Instead of creating the default cluster input, copy a custom control from the Smith Plot example VIs in the `examples\picture\demos.llb` and place it on the front panel.

If you are graphing load impedances, you can represent an impedance as a complex number of the form $r + jx$.

To avoid losing detail in the Smith plot, use the Normalize Smith Plot VI to normalize the data. You can pass the data you normalize with the Normalize Smith Plot VI directly to the Smith Plot VI. You usually scale Smith plot data with respect to the characteristic impedance (Z_0) of the system.

Picture Functions VIs

Use the Picture Functions VIs to draw shapes and enter text into a picture indicator. You can draw points, lines, shapes, and pixmaps. Pixmaps of unflattened data are 2D arrays of color, where each value corresponds to a color or to an index into an array of RGB color values, depending on the color depth.

The first row of the **Picture Functions** palette contains VIs you use to draw points and lines. A point is a cluster of two 16-bit signed integers that represent the x- and y-coordinates of a pixel.

When you use the Picture Functions VIs, the picture remembers the position of the graphics pen. For most of the Picture Functions VIs, you must specify absolute coordinates—that is, relative to the origin (0, 0). With the Draw Line VI and the Move Pen VI, you can specify either absolute or relative coordinates. Relative coordinates are relative to the current location of the pen. You can use the Move Pen VI to change the location of the pen without drawing. Only the Draw Point VI, Move Pen VI, Draw Line VI, and Draw Multiple Lines VI change the location of the pen.

The second row of the **Picture Functions** palette contains VIs you use to draw a shape. Each of these VIs draws a shape in a rectangular area of a picture. You specify a rectangle as a cluster of four values that represent the left, top, right, and bottom pixels.

The third row of the **Picture Functions** palette contains VIs you use to draw text in a picture. The Get Text Rect VI does not draw any text. Instead, you use it to calculate the size of a bounding rectangle of a string.

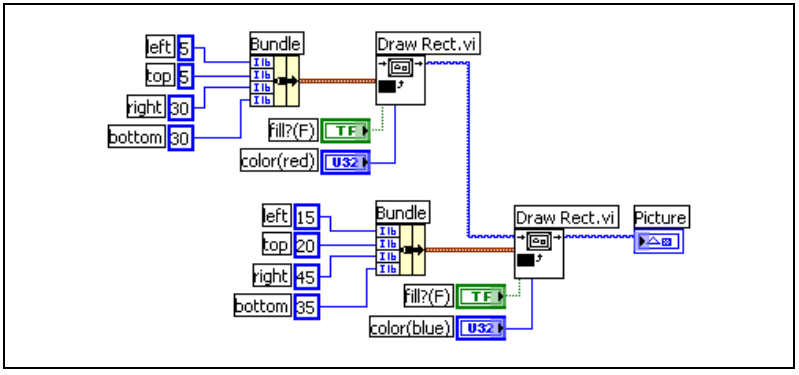
The fourth row of the **Picture Functions** palette contains VIs you use to draw flattened and unflattened pixmaps in a picture, to apply a mask to an image, to obtain a subset of a source image, and to convert a picture data type to a flattened image data cluster.

The last row of the Picture Functions palette contains the empty picture constant, which you use when you need to start with or make changes to an empty picture. The last row of the palette also contains VIs you use to convert red, green, and blue values to the corresponding RGB color and to convert a color to its respective red, green, and blue components.

You can wire the pictures you create with the Picture Functions VIs only to a picture indicator or to the **picture** input of a Picture Functions VI. LabVIEW draws the picture when it updates the picture indicator on an open front panel.

Each Picture Functions VI concatenates its drawing instruction to the drawing instructions wired to the **picture** input and returns the concatenated drawing instructions in the **new picture** output.

The following block diagram uses the Draw Rect VI to draw two overlapping rectangles.



Creating and Modifying Colors with the Picture Functions VIs

Many of the Picture Functions VIs have a **color** input to modify the color of the shapes and text. The easiest way to specify a color is to use a color box constant and click the constant to select a color.

To create colors as the result of calculations rather than with color box constants, you need to understand how a color box specifies a color using a numeric value.

A 32-bit signed integer represents a color, and the lower three bytes represent the red, green, and blue components of the color. For a range of blue colors, create an array of 32-bit integers where the blue values of each element change and are greater than the red and green values. To create a range of gray colors, create an array of 32-bit integers where the red, green, and blue values of each element are the same.

Graphics Formats VIs

Use the Graphics Formats VIs to read data from and write data to several standard graphics file formats, including bitmap (`.bmp`), Portable Network Graphics (`.png`), and Joint Photographic Experts Group (`.jpg`) files. You can use these VIs to perform the following tasks:

- Read data from a graphics file for display in a picture indicator.
- Read data for image manipulation.
- Write an image for viewing in other applications.

Bitmap data are 2D arrays in which each point varies depending on the color depth. For example, in a black-and-white, or 1-bit, image, each point is Boolean. In 4-bit and 8-bit images, each point is an index in a color table. For 24-bit true-color images, each point is a mixture of red, green, and blue (RGB) values.

The VIs that read and write graphics files work with data in a simple, flattened format that is closer to the way graphics files are written to disk, with the data stored in a 1D array. These graphics files are pixmaps, which are similar in concept to bitmaps. You can display this flattened data directly using the Draw Flattened Pixmap VI.

To manipulate the data as a 2D array, you can convert it to the appropriate format using the Unflatten Pixmap VI and Flatten Pixmap VI.

Sound VIs

Use the Sound VIs to integrate sound files and functions into your VIs. You can use these VIs to perform the following tasks:

- Create VIs that play sound files, such as a recorded warning, when users perform certain actions.
- Create a VI that plays a sound file when the VI starts or finishes running or when you reach a certain point in the VI.
- Configure a sound input device to acquire sound data. Use the Sound Input VIs to acquire the sound data. You also can read any sound information coming through the device.
- Configure a sound output device to accept sound data from other Sound VIs. You can control the volume of the sound going through the device, play or pause the sound, and clear the sound from your system.

14

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, including the following:

- Opening and closing data files.
- Reading data from and writing data to files.
- Reading from and writing to spreadsheet-formatted files.
- Moving and renaming files and directories.
- Changing file characteristics.
- Creating, modifying, and reading a configuration file.

Use the high-level VIs to perform common I/O operations. Use the low-level VIs and functions to control each file I/O operation individually.

For more information…

Refer to the *LabVIEW Help* for more information about performing file I/O operations.

Basics of File I/O

A typical file I/O operation involves the following process.

1. Create or open a file. Indicate where an existing file resides or where you want to create a new file by specifying a path or responding to a dialog box to direct LabVIEW to the file location. After the file opens, a refnum represents the file. Refer to the *References to Objects or Applications* section of Chapter 4, *Building the Front Panel*, for more information about refnums.
2. Read from or write to the file.
3. Close the file.

Most File I/O VIs and functions perform only one step in a file I/O operation. However, some high-level File I/O VIs designed for common file I/O operations perform all three steps. Although these VIs are not always as efficient as the low-level functions, you might find them easier to use.

Choosing a File I/O Format

The File I/O VIs you use depend on the format of the files. You can read data from or write data to files in three formats—text, binary, and datalog. The format you use depends on the data you acquire or create and the applications that will access that data.

Use the following basic guidelines to determine which format to use:

- If you want to make your data available to other applications, such as Microsoft Excel, use text files because they are the most common and the most portable.
- If you need to perform random access file reads or writes or if speed and compact disk space are crucial, use binary files because they are more efficient than text files in disk space and in speed.
- If you want to manipulate complex records of data or different data types in LabVIEW, use datalog files because they are the best way to store data if you intend to access the data only from LabVIEW and you need to store complex data structures.

When to Use Text Files

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

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

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

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

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

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

Refer to the `examples\file\smplfile.llb` and `examples\file\sprdsht.llb` for examples of using file I/O with text files.

When to Use Binary Files

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

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

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

Note Text and binary files are both known as byte stream files, which means they store data as a sequence of characters or bytes.

Refer to the Read Binary File and Write Binary File VIs in the `examples\file\smplfile.llb` for examples of reading and writing an array of double-precision floating-point values to and from a binary file, respectively.

When to Use Datalog Files

Use datalog files to access and manipulate data only in LabVIEW and to store complex data structures quickly and easily.

A datalog file stores data as a sequence of identically structured records, similar to a spreadsheet, where each row represents a record. Each record in a datalog file must have the same data types associated with it. LabVIEW writes each record to the file as a cluster containing the data to store. However, the components of a datalog record can be any data type, which you determine when you create the file.

For example, you can create a datalog whose record data type is a cluster of a string and a number. Then, each record of the datalog is a cluster of a string and a number. However, the first record could be (`"abc",1`), while the second record could be (`"xyz",7`).

Using datalog files requires little manipulation, which makes writing and reading much faster. It also simplifies data retrieval because you can read the original blocks of data back as a record without having to read all records that precede it in the file. Random access is fast and easy with datalog files because all you need to access the record is the record number. LabVIEW sequentially assigns the record number to each record when it creates the datalog file.

You can access datalog files from the front panel and from the block diagram. Refer to the *Logging Front Panel Data* section of this chapter for more information about accessing datalog files from the front panel.

LabVIEW writes a record to a datalog file each time the associated VI runs. You cannot overwrite a record after LabVIEW writes it to a datalog file. When you read a datalog file, you can read one or more records at a time.

Refer to the `examples\file\datalog.llb` for examples of reading and writing datalog files.

Using High-Level File I/O VIs

Use the high-level File I/O VIs to perform common I/O operations, such as writing to or reading from the following types of data:

- Characters to or from text files.
- Lines from text files.
- 1D or 2D arrays of single-precision numerics to or from spreadsheet text files.
- 1D or 2D arrays of single-precision numerics or 16-bit signed integers to or from binary files.

You can save time and programming effort by using the high-level VIs to write to and read from files. The high-level VIs perform read or write operations in addition to opening and closing the file. Avoid placing the high-level VIs in loops, because the VIs perform open and close operations each time they run. Refer to the *Disk Streaming* section of this chapter for more information about keeping files open while you perform multiple operations.

The high-level VIs expect a file path input. If you do not wire a file path, a dialog box appears for you to specify a file to read from or write to. If an error occurs, the high-level VIs display a dialog box that describes the error. You can choose to halt execution or to continue.

Figure 14-1 shows how to use the high-level Write To Spreadsheet File VI to send numbers to a Microsoft Excel spreadsheet file. When you run this VI, LabVIEW prompts you to write the data to an existing file or to create a new file.

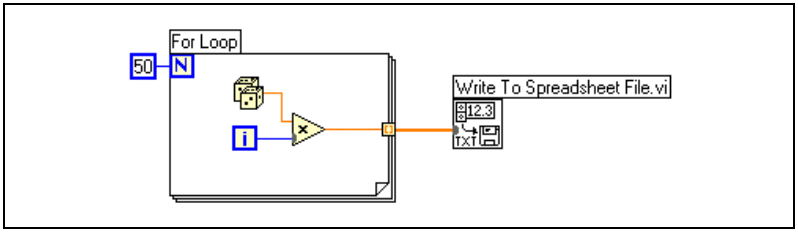


Figure 14-1. Using a High-Level VI to Write to a Spreadsheet

Use the Binary File VIs to read from and write to files in binary format. Data can be integers or single-precision floating-point numerics.

Using Low-Level and Advanced File I/O VIs and Functions

Use low-level File I/O VIs and functions and the Advanced File I/O functions to control each file I/O operation individually.

Use the principal low-level functions to create or open a file, write data to or read data from the file, and close the file. Use the other low-level functions to perform the following tasks:

- Create directories.
- Move, copy, or delete files.
- List directory contents.
- Change file characteristics.
- Manipulate paths.

A path, shown at left, is a LabVIEW data type that identifies the location of a file on disk. The path describes the volume that contains the file, the directories between the top-level of the file system and the file, and the name of the file. Enter or display a path using the standard syntax for a given platform with the path control or indicator. Refer to the *Path Controls and Indicators* section of Chapter 4, *Building the Front Panel*, for more information about path controls and indicators.

Figure 14-2 shows how to use low-level VIs and functions to send numbers to a Microsoft Excel spreadsheet file. When you run this VI, the Open/Create/Replace VI opens the `numbers.xls` file. The Write File function writes the string of numbers to the file. The Close function closes the file. If you do not close the file, the file stays in memory and is not accessible from other applications or to other users.

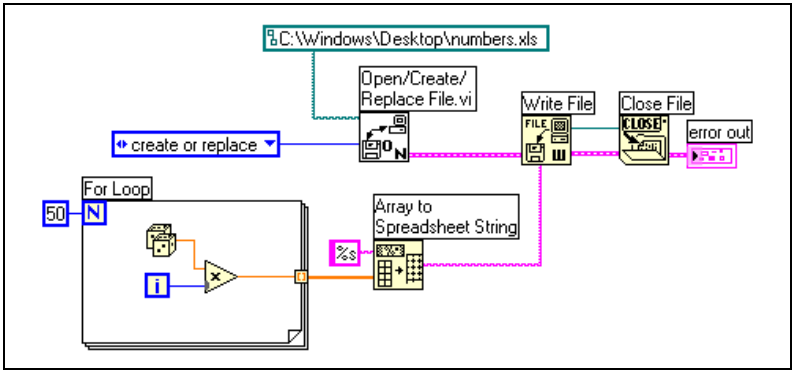


Figure 14-2. Using a Low-Level VI to Write to a Spreadsheet

Compare the VI in Figure 14-2 to the VI in Figure 14-1, which completes the same task. In Figure 14-2, you have to use the Array To Spreadsheet String function to format the array of numbers as a string. The Write To Spreadsheet File VI in Figure 14-1 opens the file, converts the array of numbers to a string, and closes the file.

Refer to the Write Datalog File Example VI in the `examples\file\datalog.llb` for an example of using low-level File I/O VIs and functions.

Disk Streaming

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

Disk streaming reduces the number of times the function interacts with the operating system to open and close the file. To create a typical disk-streaming operation, place the Open/Create/Replace File VI before the loop and the Close File function after the loop. Continuous writing to a file then can occur within the loop without the overhead associated with opening and closing the file.

Disk streaming is ideal in lengthy data acquisition operations where speed is critical. You can write data continuously to a file while acquisition is still in progress. For best results, avoid running other VIs and functions, such as Analyze VIs and functions, until you complete the acquisition.

Creating Text and Spreadsheet Files

To write data to a text file, you must convert your data to a string. To write data to a spreadsheet file, you must format the string as a spreadsheet string, which is a string that includes delimiters, such as tabs. Refer to the *Formatting Strings* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about formatting strings.

Writing text to text files requires no formatting because most word processing applications that read text do not require formatted text. To write a text string to a text file, use the Write Characters To File VI, which automatically opens and closes the file.

Use the Write To Spreadsheet File VI or the Array To Spreadsheet String function to convert a set of numbers from a graph, a chart, or an acquisition into a spreadsheet string. Refer to the *Using High-Level File I/O VIs* and *Using Low-Level and Advanced File I/O VIs and Functions* sections of this chapter for more information about using these VIs and functions.

Reading text from a word processing application might result in errors because word processing applications format text with different fonts, colors, styles, and sizes that the high-level File I/O VIs cannot process.

If you want to write numbers and text to a spreadsheet or word processing application, use the String functions and the Array functions to format the data and to combine the strings. Then write the data to a file. Refer to Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about using these functions to format and combine data.

Formatting and Writing Data to Files

Use the Format Into File function to format string, numeric, path, and Boolean data as text and to write the formatted text to a file. Often you can use this function instead of separately formatting the string with the Format Into String function and writing the resulting string with the Write Characters To File VI or Write File function.

Refer to the *Formatting Strings* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about formatting strings.

Scanning Data from Files

Use the Scan From File function to scan text in a file for string, numeric, path, and Boolean values and then convert the text into a data type. Often

you can use this function instead of reading data from a file with the Read File function or Read Characters From File VI and scanning the resulting string with the Scan From String function.

Creating Binary Files

To create a binary file, acquire a set of numbers and write them to a file. Use the Write To I16 and Write To SGL VIs to save 1D and 2D arrays of 16-bit integers or single-precision floating-point numbers to a file. Use the Read I16 and Read SGL VIs to read the files you created.

To write numbers of different data types, such as double-precision floating-point numerics or 32-bit unsigned integers, use the low-level or Advanced File functions. When you read the file, use the Read File function and specify the data type of the number.

Refer to the Read Binary File and Write Binary File VIs in the `examples\file\smplfile.llb` for examples of writing and reading floating-point numerics to and from a binary file.

Creating Datalog Files

You can create and read datalog files by enabling front panel datalogging or by using the low-level or Advanced File functions to acquire data and write the data to a file. Refer to the *Logging Front Panel Data* section of this chapter for more information about creating and accessing datalog files from the front panel.

You do not have to format the data in a datalog file. However, when you write or read datalog files, you must specify the data type. For example, if you acquire a temperature reading with the time and date the temperature was recorded, you write the data to a datalog file and specify the data as a cluster of one number and two strings. Refer to the Simple Temp Datalogger VI in the `examples\file\datalog.llb` for an example of writing data to a datalog file.

If you read a file that includes a temperature reading with the time and date the temperature was recorded, you specify that you want to read a cluster of one number and two strings. Refer to the Simple Temp Datalog Reader VI in the `examples\file\datalog.llb` for an example of reading a datalog file.

Writing Waveforms to Files

Use the Write Waveforms to File and Export Waveforms to Spreadsheet File VIs to send waveforms to files. You can write waveforms to spreadsheet, text, or datalog files.

If you expect to use the waveform you create only in a VI, save the waveform as a datalog file (`.log`). Refer to the *When to Use Datalog Files* section of this chapter for more information about datalogging.

The VI in Figure 14-3 acquires multiple waveforms, displays them on a graph, and writes them to a spreadsheet file.

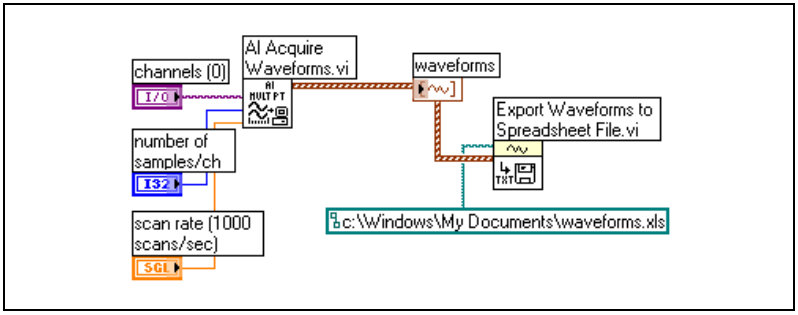


Figure 14-3. Writing Multiple Waveforms to a Spreadsheet File

Reading Waveforms from Files

Use the Read Waveform from File VI to read waveforms from a file. After you read a single waveform, you can add or edit waveform data type components with the Build Waveform function, or you can extract waveform components with the Get Waveform Attribute function.

The VI in Figure 14-4 reads a waveform from a file, edits the **dt** component of the waveform, and plots the edited waveform to a graph.

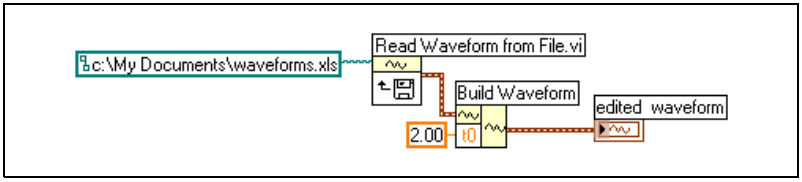


Figure 14-4. Reading a Waveform from a File

The Read Waveform from File VI also reads multiple waveforms from a file. The VI returns an array of waveform data types, which you can display in a multiplot graph. If you want to access a single waveform from a file, you must index the array of waveform data types, as shown in Figure 14-5. Refer to the *Arrays* section of Chapter 10, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about indexing arrays. The VI accesses a file that includes multiple waveforms. The Index Array function reads the first and third waveforms in the file and plots them on two separate waveform graphs. Refer to Chapter 8, *Loops and Structures*, for more information about graphs.

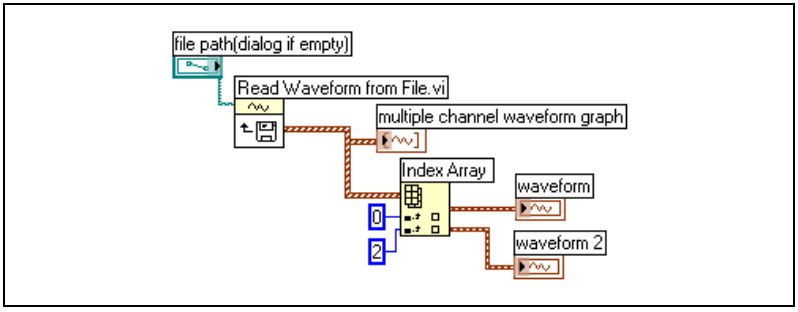


Figure 14-5. Reading Multiple Waveforms from a File

Flow-Through Parameters

Many File I/O VIs and functions contain flow-through parameters, typically a refnum or path, that return the same value as the corresponding input parameter. Use these parameters to control the execution order of the functions. By wiring the flow-through output of the first node you want to execute to the corresponding input of the next node you want to execute, you create artificial data dependency. Without these flow-through parameters, you have to use sequence structures to ensure that file I/O operations take place in the order you want. Refer to the *Data Dependency and Artificial Data Dependency* section of Chapter 5, *Building the Block Diagram*, for more information about artificial data dependency.

Creating Configuration Files

Use the Configuration File VIs to read and create standard Windows configuration settings (`.ini`) files and to write platform-specific data, such as paths, in a platform-independent format so that you can use the files these VIs generate across multiple platforms. The Configuration File VIs do not use a standard file format for configuration files. While you can use the Configuration File VIs on any platform to read and write files created by the VIs, you cannot use the Configuration File VIs to create or modify configuration files in a Mac OS or UNIX format.

Refer to the `examples\file\config.llb` for examples of using the Configuration File VIs.

Note The standard extension for Windows configuration settings files is `.ini`, but the Configuration File VIs work with files with any extension, provided the content is in the correct format. Refer to the *Windows Configuration Settings File Format* section of this chapter for more information about configuring the content.

Using Configuration Settings Files

A standard Windows configuration settings file is a specific format for storing data in a text file. You can programmatically access data within the `.ini` file easily because it follows a specific format.

For example, consider a configuration settings file with the following contents:

```
[Data]
Value=7.2
```

You can use the Configuration File VIs to read this data, as shown in Figure 14-6. This VI uses the Read Key VI to read the **key** named `Value` from the **section** called `Data`. This VI works regardless of how the file changes, provided the file remains in the Windows configuration settings file format.

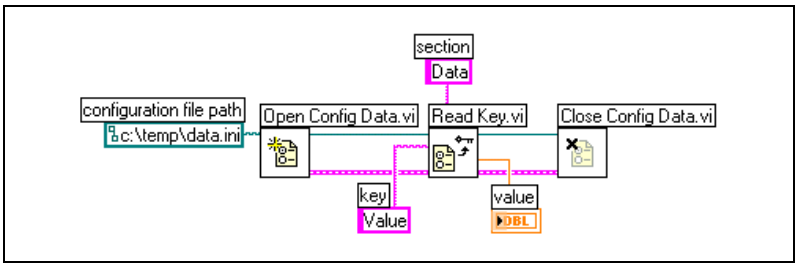


Figure 14-6. Reading Data from an .ini File

Windows Configuration Settings File Format

Windows configuration settings files are text files divided into named sections. Brackets enclose each section name. Every section name in a file must be unique. The sections contain key/value pairs separated by an equal sign (=). Within each section, every key name must be unique. The key name represents a configuration preference, and the value name represents the setting for that preference. The following example shows the arrangement of the file:

```
[Section 1]
key1=value
key2=value

[Section 2]
key1=value
key2=value
```

Use the following data types with Configuration File VIs for the value portion of the **key** parameter:

- String
- Path
- Boolean
- 64-bit double-precision floating-point numeric
- 32-bit signed integer
- 32-bit unsigned integer

The Configuration File VIs can read and write raw or escaped string data. The VIs read and write raw data byte-for-byte, without converting the data to ASCII. In converted, or escaped, strings LabVIEW stores any non-displayable text characters in the configuration settings file with the equivalent hexadecimal escape codes, such as `\0D` for a carriage return. In addition, LabVIEW stores backslash characters in the configuration

settings file as double backslashes, such as `\\` for `\`. Set the **read raw string?** or **write raw string?** inputs of the Configuration File VIs to TRUE for raw data and to FALSE for escaped data.

When VIs write to a configuration file, they place quotation marks around any string or path data that contain a space character. If a string contains quotation marks, LabVIEW stores them as `\"`. If you read and/or write to configuration files using a text editor, you might notice that LabVIEW replaced quotation marks with `\"`.

LabVIEW stores path data in a platform-independent format, the standard UNIX format for paths, in `.ini` files. The VIs interpret the absolute path `/c/temp/data.dat` stored in a configuration settings file as follows:

- **(Windows)** `c:\temp\data.dat`
- **(Mac OS)** `c:temp:data.dat`
- **(UNIX)** `/c/temp/data.dat`

The VIs interpret the relative path `temp/data.dat` as follows:

- **(Windows)** `temp\data.dat`
- **(Mac OS)** `:temp:data.dat`
- **(UNIX)** `temp/data.dat`

Logging Front Panel Data

Use front panel datalogging to record data for use in other VIs and in reports. For example, you can log data from a graph and use that data in another graph in a separate VI.

Each time a VI runs, front panel datalogging saves the front panel data to a separate datalog file, which is in the format of delimited text. You can retrieve the data in the following ways:

- Use the same VI from which you recorded the data to retrieve the data interactively.
- Use the VI as a subVI to retrieve the data programmatically.
- Use the File I/O VIs and functions to retrieve the data.

Each VI maintains a log-file binding that records the location of the datalog file where LabVIEW maintains the logged front panel data. Log-file binding is the association between a VI and the datalog file to which you log the VI data.

A datalog file contains records that include a time stamp and the data from each time you ran the VI. When you access a datalog file, you select which record you want by running the VI in retrieval mode and using the front panel controls to view the data. When you run the VI in retrieval mode, a numeric control appears at the top of the front panel, so you can navigate among the records. Refer to Figure 14-7 for an example of this numeric control.

Automatic and Interactive Front Panel Datalogging

Select **Operate»Log at Completion** to enable automatic logging. The first time you log front panel data for a VI, LabVIEW prompts you to name the datalog file. LabVIEW logs data each time you run the VI and appends a new record to the datalog file each additional time you run the VI. You cannot overwrite a record after LabVIEW writes it to a datalog file.

To log your data interactively, select **Operate»Data Logging»Log**. LabVIEW appends the data to the datalog file immediately. Log your data interactively so you can select when to log the data. Logging your data automatically logs the data each time you run the VI.

Note A waveform chart logs only one data point at a time with front panel datalogging. If you wire an array to the chart indicator, the datalog file contains a subset of the array the chart displays.

Viewing the Logged Front Panel Data Interactively

After you log data, you can view it interactively by selecting **Operate» Data Logging»Retrieve**. The data retrieval toolbar appears, as shown in Figure 14-7.

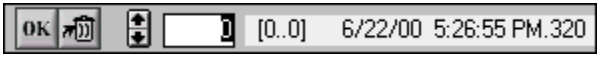


Figure 14-7. Data Retrieval Toolbar

The highlighted number indicates the data record you are viewing. The numbers in square brackets indicate the range of records you logged for the current VI. You log a record each time you run the VI. The date and time indicate when you logged the selected record. View the next or previous record by clicking the increment or decrement arrows. You also can use the up and down arrow keys on your keyboard.

In addition to the data retrieval toolbar, the front panel appearance changes according to the record you select on the toolbar. For example, when you

click the increment arrow and advance to another record, the controls and indicator display the data for that particular record at the time you logged the data. Click the **OK** button to exit retrieval mode and return to the VI whose datalog file you were viewing.

Deleting a Record

While in retrieval mode, you can delete specific records. Mark for deletion an individual record in retrieval mode by viewing that record and clicking the **Trash** button. If you click the **Trash** button again, the record is no longer marked for deletion.

Select **Operate»Data Logging»Purge Data** while in retrieval mode to delete all the records you marked for deletion.

If you do not delete your marked record before you click the **OK** button, LabVIEW prompts you to delete the marked records.

Clearing the Log-File Binding

Use log-file binding to associate a VI with the datalog file to use when logging or retrieving front panel data. You can have two or more datalog files associated with one VI. This might help you test or compare the VI data. For example, you could compare the data logged the first time you run the VI to the data logged the second time you run the VI. To associate more than one datalog file with a VI, you must clear the log-file binding by selecting **Operate»Data Logging»Clear Log File Binding**. LabVIEW prompts you to specify a datalog file the next time you run the VI either with automatic logging enabled or when you choose to log data interactively.

Changing the Log-File Binding

Change the log-file binding to log front panel data to or retrieve front panel data from a different log file by selecting **Operate»Data Logging»Change Log File Binding**. LabVIEW prompts you to select a different log file or to create a new one. You might change log-file binding when you want to retrieve different data into a VI or append data from the VI to another datalog file.

Retrieving Front Panel Data Programmatically

You also can retrieve logged data using a subVI or using the File I/O VIs and functions.

Retrieving Front Panel Data Using a SubVI

When you right-click a subVI and select **Enable Database Access** from the shortcut menu, a yellow box appears around the subVI, as shown in Figure 14-8.

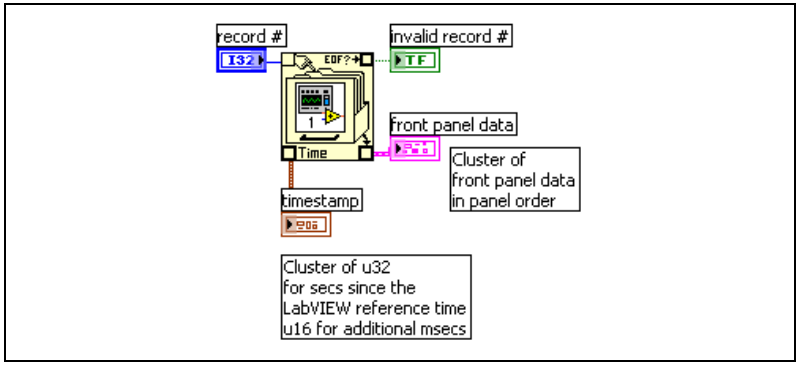


Figure 14-8. Retrieving Logged Data

The yellow box that looks like a filing cabinet includes terminals for accessing data from the datalog file. When you enable database access for the subVI, the inputs and outputs of the subVI actually act as outputs, returning their logged data. **record #** indicates the record to retrieve, **invalid record #** indicates whether the record number exists, **timestamp** is the time the record was created, and **front panel data** is a cluster of the front panel objects. You can access the data of a front panel object by wiring the **front panel data** cluster to the Unbundle function.

You also can retrieve values for specific inputs and outputs by wiring directly to the corresponding terminal on the subVI, as shown in Figure 14-9.

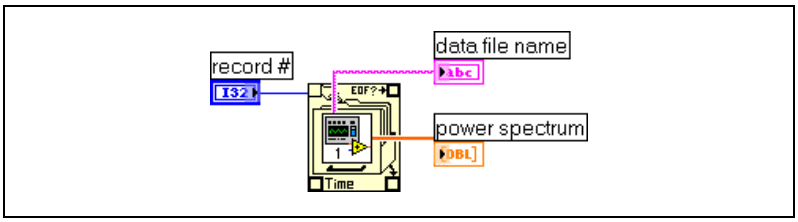


Figure 14-9. Retrieving Logged Data through SubVI Terminals

If you run the VI, the subVI does not run. Instead, it returns the logged data from its front panel to the VI front panel as a cluster.

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

Specifying Records

The subVI has *n* logged records, and you can wire any number from –*n* to *n* – 1 to the **record #** terminal of the subVI. You can access records relative to the first logged record using non-negative record numbers. 0 represents the first record, 1 represents the second record, and so on, through *n* – 1, which represents the last record.

You can access records relative to the last logged record using negative record numbers. –1 represents the last record, –2 represents the second to the last, and so on, through –*n*, which represents the first record. If you wire a number outside the range –*n* to *n* – 1 to the **record #** terminal, the **invalid record #** output is TRUE, and the subVI retrieves no data.

Retrieving Front Panel Data Using File I/O Functions

You also can retrieve data you logged from the front panel by using File I/O VIs and functions, such as the Read File function. The data type of each record in the front panel datalog file creates two clusters. One cluster contains a time stamp, and the other cluster contains the front panel data. The time stamp cluster includes an 32-bit unsigned integer that represents seconds and an 16-bit unsigned integer that represents milliseconds elapsed since the LabVIEW system time, which is 12:00 a.m., January 1, 1904, Universal time.

You access the records of front panel datalog files with the same File I/O functions you use to access datalog files you created programmatically. Enter the **datalog record type** as the type input to the File Open function, as shown in Figure 14-10.

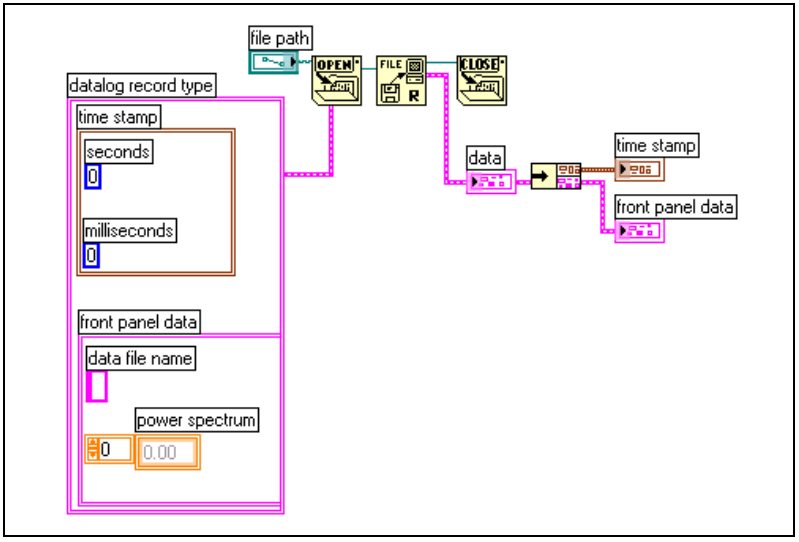


Figure 14-10. Retrieving Logged Data Using the File Open Function

LabVIEW Data Directory

Use the default `LabVIEW Data` directory to store the data files LabVIEW generates, such as `.lvm` 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 `.lvm` files it generates in this directory, and the Read LabVIEW Measurement File Express VI reads from this directory. The Default Data Directory constant, shown at left, and the Default Data Directory property also return the `LabVIEW Data` directory by default.

Select **Tools»Options** and select **Paths** from the top pull-down menu 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.

LabVIEW Measurement Data File

The LabVIEW measurement data file (`.lvm`) includes data the Write LabVIEW Measurement File Express VI generates. The LabVIEW data file is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. In addition to the data an Express VI generates, the `.lvm` file includes information about the data, such as the date and time the data was generated.

Refer to the *Getting Started with LabVIEW* manual for more information about saving and retrieving data using Express VIs.

You can use a LabVIEW measurement data file with NI DIAdem.

15

Documenting and Printing VIs

You can use LabVIEW to document and print VIs.

The *Documenting VIs* section of this chapter describes how to record information about the block diagram and/or the front panel at any stage of development in printed documentation about VIs.

The *Printing VIs* section of this chapter describes options for printing VIs. Some options are more appropriate for printing information about VIs, and others are more appropriate for reporting the data and results the VIs generate. Several factors affect which printing method you use, including if you want to print manually or programmatically, how many options you need for the report format, if you need the functionality in the stand-alone applications you build, and on which platforms you run the VIs.

For more information...

Refer to the *LabVIEW Help* for more information about documenting and printing VIs and generating reports.

Documenting VIs

You can use LabVIEW to track development, document a finished VI, and create instructions for users of VIs. You can view documentation within LabVIEW, print it, and save it to HTML, RTF, or text files.

To create effective documentation for VIs, add comments to the VI revision history and create VI and object descriptions.

Setting up the VI Revision History

Use the **History** window to display the development history of a VI, including revision numbers. As you make changes to the VI, record and track them in the **History** window. Select **Tools»VI Revision History** to display the **History** window. You also can print the revision history or save it to an HTML, RTF, or text file. Refer to the *Printing Documentation*

section of this chapter for more information about printing the revision history or saving it to a file.

Revision Numbers

The revision number is an easy way to track changes to a VI. The revision number starts at zero and increases incrementally every time you save the VI. To display the current revision number in the title bar of the VI and the title bar of the **History** window, select **Tools»Options**, select **Revision History** from the top pull-down menu, and place a checkmark in the **Show revision number in titlebar** checkbox.

The number LabVIEW displays in the **History** window is the next revision number, which is the current revision number plus one. When you add a comment to the history, the header of the comment includes the next revision number. The revision number does not increase when you save a VI if you change only the history.

Revision numbers are independent of comments in the **History** window. Gaps in revision numbers between comments indicate that you saved the VI without a comment.

Because the history is strictly a development tool, LabVIEW automatically removes the history when you remove the block diagram of a VI. Refer to the *Distributing VIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about removing the block diagram. The **History** window is not available in the run-time version of a VI. The **General** page of the **VI Properties** dialog box displays the revision number, even for VIs without block diagrams. Click the **Reset** button in the **History** window to erase the revision history and reset the revision number to zero.

Creating VI and Object Descriptions

Create descriptions for VIs and their objects, such as controls and indicators, to describe the purpose of the VI or object and to give users instructions for using the VI or object. You can view the descriptions in LabVIEW, print them, or save them to HTML, RTF, or text files.

Create, edit, and view VI descriptions by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu. Create, edit, and view object descriptions by right-clicking the object and selecting **Description and Tip** from the shortcut menu. Tip strips are brief descriptions that appear when you move the cursor over an object while a VI runs. If you do not enter a tip in the **Description and Tip** dialog box, no tip strip appears. The VI or object description appears in the **Context**

Help window when you move the cursor over the VI icon or object, respectively.

Printing Documentation

Select **File»Print** to print VI documentation or save it to HTML, RTF, or text files. You can select a built-in format or create a custom format for documentation. The documentation you create can include the following items:

- Icon and connector pane
- Front panel and block diagram
- Controls, indicators, and data type terminals
- VI and object descriptions
- VI hierarchy
- List of subVIs
- Revision history

Note The documentation you create for certain types of VIs cannot include all the previous items. For example, a polymorphic VI does not have a front panel or a block diagram, so you cannot include those items in the documentation you create for a polymorphic VI.

Saving Documentation to HTML, RTF, or Text Files

You can save VI documentation to HTML, RTF, or text files. You can import HTML and RTF files into most word processing applications, and you can use HTML and RTF files to create compiled help files. You also can use the HTML files LabVIEW generates to display VI documentation on the Web. Refer to the *Creating Your Own Help Files* section of this chapter for more information about using HTML and RTF files to create help files. Refer to the *Printing VIs Programmatically* section of this chapter for more information about printing and saving documentation to HTML, RTF, and text files programmatically.

When you save documentation to an RTF file, specify if you want to create a file suitable for online help files or for word processing. In the help file format, LabVIEW saves the graphics to external bitmap files. In the word processing file format, LabVIEW embeds the graphics in the document. For HTML files, LabVIEW saves all graphics externally in the JPEG, PNG, or GIF formats.

Selecting Graphic Formats for HTML Files

When you save documentation to an HTML file, you can select the format of the graphics files and the color depth.

The JPEG format compresses graphics well but can lose some graphic detail. This format works best for photos. For line art, front panels, and block diagrams, JPEG compression can result in fuzzy graphics and uneven colors. JPEG graphics are always 24-bit graphics. If you select a lower color depth such as black-and-white, graphics save with the depth you requested, but the result is still a 24-bit graphic.

The PNG format also compresses graphics well, although not always as well as the JPEG format. However, PNG compression does not lose any detail. Also, it supports 1-bit, 4-bit, 8-bit, and 24-bit graphics. For lower bit depth, the resulting graphic compresses much better than JPEG. The PNG format replaces the Graphics Interchange Format (GIF). Although the PNG format has advantages over both JPEG and GIF, it is not as well supported by Web browsers.

The GIF format also compresses graphics well and is supported by most Web browsers. Because of licensing issues, LabVIEW does not save graphics as compressed GIF files but might in the future. Use a graphics format converter to convert the uncompressed GIF files that LabVIEW saves to compressed GIF files. For higher quality compressed GIF files, select the PNG format when you save the documentation and use a graphics format converter to convert the PNG files that LabVIEW saves to GIF files. Starting with the PNG format produces higher quality graphics because the PNG format is an exact reproduction of the original graphic. Modify the HTML file that LabVIEW generated to refer to the GIF files with the `.gif` extension.

Naming Conventions for Graphic Files

When you generate HTML or RTF documentation with external graphics, LabVIEW saves the control and indicator data type terminals to graphic files with consistent names. If a VI has multiple terminals of the same type, LabVIEW creates only one graphic file for that type. For example, if a VI has three 32-bit signed integer inputs, LabVIEW creates a single `ci32.`*x* file, where *x* is the extension corresponding to the graphic format.

Creating Your Own Help Files

You can use the HTML or RTF files LabVIEW generates to create your own compiled help files. **(Windows)** You can compile the individual HTML files LabVIEW generates into an HTML Help file.

You can compile the RTF files LabVIEW generates into a **(Windows)** WinHelp, **(Mac OS)** QuickHelp, or **(UNIX)** HyperHelp file.

Create links from VIs to HTML files or compiled help files by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu.

Printing VIs

You can use the following primary methods to print VIs:

- Select **File»Print Window** to print the contents of the active window.
- Select **File»Print** to print more comprehensive information about a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on. Refer to the *Printing Documentation* section of this chapter for more information about using this method to print VIs.
- Use the VI Server to programmatically print any VI window or VI documentation at any time. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using this method to print VIs.

Printing the Active Window

Select **File»Print Window** to print the contents of the active front panel or block diagram window with the minimum number of prompts. LabVIEW prints the workspace of the active window, including any objects not in the visible portion of the window. LabVIEW does not print the title bar, menu bar, toolbar, or scrollbars.

Select **File»VI Properties** and select **Print Options** from the **Category** pull-down menu to configure how LabVIEW prints a VI when you select **File»Print Window** or when you print programmatically. Refer to the *Printing VIs Programmatically* section of this chapter for more information about printing programmatically.

Printing VIs Programmatically

Use any of the following methods to print VIs programmatically rather than interactively with the dialog boxes that appear when you select **File»Print Window** and **File»Print**:

- Set a VI to automatically print its front panel every time it finishes running.
- Create a subVI to print the VI.
- Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns.
- Use the VI Server to programmatically print a VI window or to print VI documentation or save it to HTML, RTF, or text files at any time. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using this method to print VIs.

Note If you print VI documentation from a built application, you can print only the front panels.

Printing the Front Panel of a VI after the VI Runs

Select **Operate»Print at Completion** to print the front panel of a VI when it finishes running. You also can select **File»VI Properties**, select **Print Options** from the **Category** pull-down menu, and place a checkmark in the **Automatically Print Panel Every Time VI Completes Execution** checkbox.

Selecting these options is similar to selecting **File»Print Window** when the front panel is the active window.

If you use the VI as a subVI, LabVIEW prints when that subVI finishes running and before the subVI returns to the caller.

Using a SubVI to Print Data from a Higher Level VI

In some cases, you might not want a VI to print every time it finishes running. You might want printing to occur only if the user clicks a button or if some condition occurs, such as a test failure. You also might want more control over the format for the printout, or you might want to print only a subset of the controls. In these cases, you can use a subVI that is set to print at completion.

Create a subVI and format the front panel the way you want LabVIEW to print it. Instead of selecting **Operate»Print at Completion** in the higher level VI, select it in the subVI. When you want to print, call the subVI and wire the data you want to print to the subVI.

Generating and Printing Reports

Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns. Use the Easy Print VI Panel or Documentation VI to generate a basic report that contains VI documentation. Use the Easy Text Report VI to generate a basic report that contains data the VI returns. Use the other Report Generation VIs to generate more complex reports.

Note You can generate reports only in the LabVIEW Full and Professional Development Systems.

Use the Report Generation VIs to perform the following tasks:

- Append text, graphics, tables, or VI documentation to a report.
- Set text font, size, style, and color.
- Set the report orientation—portrait or landscape.
- Set the report headers and footers.
- Set margins and tabs.

Additional Printing Techniques

If standard LabVIEW printing methods do not meet your needs, you can use the following additional techniques:

- Print data on a line-by-line basis. If you have a line-based printer connected to a serial port, use the Serial Compatibility VIs to send text to the printer. Doing so generally requires some knowledge of the command language of the printer.
- Export data to other applications, such as Microsoft Excel, save the data to a file, and print from the other application.
- **(Windows, Mac OS X, and UNIX)** Use the System Exec VI.
- **(Mac OS)** Use the AESend Print Document VI.
- **(Windows)** Use ActiveX to make another application print data. Refer to Chapter 19, *Windows Connectivity*, for more information about ActiveX.

16

Customizing VIs

You can configure VIs and subVIs to work according to your application needs. For example, if you plan to use a VI as a subVI that requires user input, configure the VI so that its front panel appears each time you call it.

You can configure a VI in many ways, either within the VI itself or programmatically by using the VI Server. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using the VI Server to configure how a VI behaves.

For more information...

Refer to the *LabVIEW Help* for more information about customizing VIs.

Configuring the Appearance and Behavior of VIs

Select **File»VI Properties** to configure the appearance and behavior of a VI. Use the **Category** pull-down menu at the top of the dialog box to select from several different option categories, including the following:

- **General**—Displays the current path where a VI is saved, its revision number, revision history, and any changes made since the VI was last saved. You also can use this page to edit the icon or the size of the alignment grid for the VI.
- **Documentation**—Use this page to add a description of the VI and link to a help file topic. Refer to the *Documenting VIs* section of Chapter 15, *Documenting and Printing VIs*, for more information about the documentation options.
- **Security**—Use this page to lock or password-protect a VI.
- **Window Appearance**—Use this page to configure various window settings.
- **Window Size**—Use this page to set the size of the window.
- **Execution**—Use this page to configure how a VI runs. For example, you can configure a VI to run immediately when it opens or to pause when called as a subVI. You also can configure the VI to run at

different priorities. For example, if it is crucial that a VI runs without waiting for another operation to complete, configure the VI to run at time-critical (highest) priority. Refer to the *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note for more information about creating multithreaded VIs.

- **Editor Options**—Use this page to set the size of the alignment grid for the current VI and to change the style of control or indicator LabVIEW creates when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu. Refer to the *Aligning and Distributing Objects* section of Chapter 4, *Building the Front Panel*, for more information about the alignment grid.

Customizing Menus

You can create custom menus for every VI you build, and you can configure VIs to show or hide menu bars. Show and hide menu bars by selecting **File»VI Properties**, selecting **Windows Appearance** from the **Category** pull-down menu, clicking the **Customize** button, and placing or removing a checkmark from the **Show Menu Bar** checkbox.

Configuring menus includes creating the menu and providing the block diagram code that executes when the user selects the various menu items.

Note Custom menus appear only while the VI runs.

Creating Menus

You can build custom menus or modify the default LabVIEW menus statically when you edit the VI or programmatically when you run the VI. When you select **Edit»Run-Time Menu** and create a menu in the **Menu Editor** dialog box, LabVIEW creates a run-time menu (`.rtm`) file so you can have a custom menu bar on a VI rather than the default menu bar. After you create and save the `.rtm` file, you must maintain the same relative path between the VI and the `.rtm` file.

Use the **Menu Editor** dialog box to associate a custom `.rtm` file with a VI. When the VI runs, it loads the menu from the `.rtm` file. You also can use the **Menu Editor** dialog box to build custom menus either with application items, which are menu items LabVIEW provides in the default menu, or with user items, which are menu items you add. LabVIEW defines the behavior of application items, but you control the behavior of user items with the block diagram. Refer to the *Menu Selection Handling* section of

this chapter for more information about handling user menu item selections.

Use the Menu Editor dialog box to customize menus when editing a VI. Use the Menu functions to customize menus programmatically at run time. The functions allow you to insert, delete, and modify the attributes of user items. You can only add or delete application items because LabVIEW defines the behavior and state of application items.

Menu Selection Handling

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

After you build a custom menu, build a Case structure on the block diagram that executes, or handles, each item in the custom menu. This process is called menu selection handling. LabVIEW handles all application items implicitly.

In Figure 16-1, the Get Menu Selection function reads the menu item the user selects and passes the menu item into the Case structure, where the menu item executes.

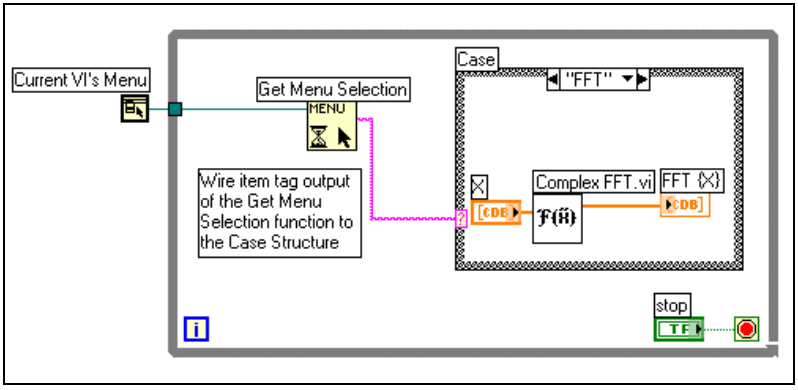


Figure 16-1. Block Diagram Using Menu Handling

If you know that a certain menu item takes a long time to process, wire a Boolean control to the **block menu** input of the Get Menu Selection function and set the Boolean control to TRUE to disable the menu bar so the user cannot select anything else on the menu while LabVIEW processes the menu item. Wire a TRUE value to the Enable Menu Tracking function to enable the menu bar after LabVIEW processes the menu item.

You also can use the Event structure to handle menu events. Refer to Chapter 9, *Event-Driven Programming*, for more information about the Event structure.

17

Programmatically Controlling VIs

You can access the VI Server through block diagrams, ActiveX technology, and the TCP protocol to communicate with VIs and other instances of LabVIEW so you can programmatically control VIs and LabVIEW. You can perform VI Server operations on a local computer or remotely across a network.

For more information…

Refer to the *LabVIEW Help* for more information about programmatically controlling VIs.

Capabilities of the VI Server

Use the VI Server to perform the following programmatic operations:

- Call a VI remotely.
- Configure an instance of LabVIEW to be a server that exports VIs you can call from other instances of LabVIEW on the Web. For example, if you have a data acquisition application that acquires and logs data at a remote site, you can sample that data occasionally from your local computer. By changing your LabVIEW preferences, you can make some VIs accessible on the Web so that transferring the latest data is as easy as a subVI call. The VI Server handles the networking details. The VI Server also works across platforms, so the client and the server can run on different platforms.
- Edit the properties of a VI and LabVIEW. For example, you can dynamically determine the location of a VI window or scroll a front panel so that a part of it is visible. You also can programmatically save any changes to disk.
- Update the properties of multiple VIs rather than manually using the **File»VI Properties** dialog box for each VI.

- Retrieve information about an instance of LabVIEW, such as the version number and edition. You also can retrieve environment information, such as the platform on which LabVIEW is running.
- Dynamically load VIs into memory when another VI needs to call them, rather than loading all subVIs when you open a VI.
- Create a plug-in architecture for the application to add functionality to the application after you distribute it to customers. For example, you might have a set of data filtering VIs, all of which take the same parameters. By designing the application to dynamically load these VIs from a plug-in directory, you can ship the application with a partial set of these VIs and make more filtering options available to users by placing the new filtering VIs in the plug-in directory.

Building VI Server Applications

The programming model for VI Server applications is based on refnums. Refnums also are used in file I/O, network connections, and other objects in LabVIEW. Refer to the *References to Objects or Applications* section of Chapter 4, *Building the Front Panel*, for more information about refnums.

Typically, you open a refnum to an instance of LabVIEW or to a VI. You then use the refnum as a parameter to other VIs. The VIs get (read) or set (write) properties, execute methods, or dynamically load a referenced VI. Finally, you close the refnum, which releases the referenced VI from memory.

Use the following Application Control functions and nodes to build VI Server applications:

- **Open Application Reference**—Opens a reference to the local or remote application you are accessing through the server or to access a remote instance of LabVIEW.
- **Open VI Reference**—Opens a reference to a VI on the local or remote computer, or to dynamically load a VI from disk.
- **Property Node**—Gets and sets VI, object, or application properties. Refer to the *Property Nodes* section of this chapter for more information about properties.
- **Invoke Node**—Invokes methods on a VI, object, or application. Refer to the *Invoke Nodes* section of this chapter for more information about methods.

- **Call By Reference Node**—Calls a dynamically loaded VI.
- **Close Reference**—Closes any open references to the VI, object, or application you accessed using the VI Server.

Application and VI References

You access VI Server functionality through references to two main classes of objects—the Application object and the VI object. After you create a reference to one of these objects, you can pass the reference to a VI or function that performs an operation on the object.

An Application refnum refers to a local or remote instance of LabVIEW. You can use Application properties and methods to change LabVIEW preferences and return system information. A VI refnum refers to a VI in an instance of LabVIEW.

With a refnum to an instance of LabVIEW, you can retrieve information about the LabVIEW environment, such as the platform on which LabVIEW is running, the version number, or a list of all VIs currently in memory. You also can set information, such as the current user name or the list of VIs exported to other instances of LabVIEW.

When you create a refnum to a VI, LabVIEW loads the VI into memory. The VI stays in memory until you close the refnum. If you have multiple refnums to a VI open at the same time, the VI stays in memory until you close all refnums to the VI. With a refnum to a VI, you can update all the properties of the VI available in the **File»VI Properties** dialog box as well as dynamic properties, such as the position of the front panel window. You also can programmatically print the VI documentation, save the VI to another location, and export and import its strings to translate into another language.

Manipulating Application and VI Settings

Use the VI Server to get and set application and VI settings by using the Property and Invoke Nodes. You can get and set many application and VI settings only through the Property and Invoke Nodes.

Refer to `examples\viserver` for examples of using the Application and VI Class properties and methods.

Property Nodes

Use the Property Node to get and set various properties on an application or VI. Select properties from the node by using the Operating tool to click the property terminal or by right-clicking the white area of the node and selecting **Properties** from the shortcut menu.

You can read or write multiple properties using a single node, however some properties are not writable. Use the Positioning tool to resize the Property Node to add new terminals. A small direction arrow to the right of the property indicates a property you read. A small direction arrow to the left of the property indicates a property you write. Right-click the property and select **Change to Read** or **Change to Write** from the shortcut menu to change the operation of the property.

The node executes from top to bottom. The Property Node does not execute if an error occurs before it executes, so always check for the possibility of errors. If an error occurs in a property, LabVIEW ignores the remaining properties and returns an error. The **error out** cluster contains information about which property caused the error.

If the Property Node opens and returns a reference to an application or VI, use the Close Reference function to close the application or VI reference. LabVIEW closes control references when that reference is no longer needed. You do not have to explicitly close control references.

Implicitly Linked Property Nodes

When you create a Property Node from a front panel object by right-clicking the object and selecting **Create»Property Node** from the shortcut menu, LabVIEW creates a Property Node on the block diagram that is implicitly linked to the front panel object. Because these Property Nodes are implicitly linked to the object from which it was created, they have no **refnum** input, and you do not need to wire the Property Node to the terminal of the front panel object or the control reference. Refer to the *Controlling Front Panel Objects* section of this chapter for more information about control references.

Invoke Nodes

Use the Invoke Node to perform actions, or methods, on an application or VI. Unlike the Property Node, a single Invoke Node executes only a single method on an application or VI. Select a method by using the Operating tool to click the method terminal or by right-clicking the white area of the node and selecting **Methods** from the shortcut menu.

The name of the method is always the first terminal in the list of parameters in the Invoke Node. If the method returns a value, the method terminal displays the return value. Otherwise, the method terminal has no value.

The Invoke Node lists the parameters from top to bottom with the name of the method at the top and the optional parameters, which are dimmed, at the bottom.

Manipulating Application Class Properties and Methods

You can get or set properties on a local or remote instance of LabVIEW, perform methods on LabVIEW, or both. Figure 17-1 shows how to display all VIs in memory on a local computer in a string array on the front panel.

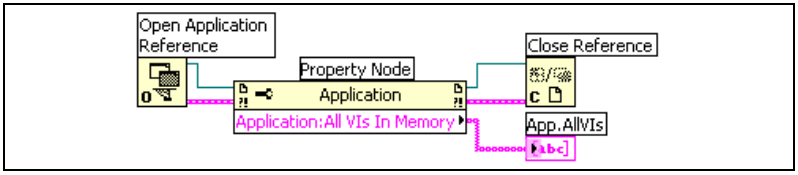


Figure 17-1. Displaying All VIs in Memory on a Local Computer

If you do not wire a refnum to the **reference** input, the Property Node or Invoke Node uses a reference to the current instance of LabVIEW. If you want to manipulate the properties or methods of another instance of LabVIEW, you must wire an application refnum to the **reference** input.

To find the VIs in memory on a remote computer, wire a string control to the **machine name** input of the Open Application Reference function, as shown in Figure 17-2, and enter the IP address or domain name. You also must select the **Exported VIs in Memory** property because the **All VIs in Memory** property used in Figure 17-1 applies only to local instances of LabVIEW.

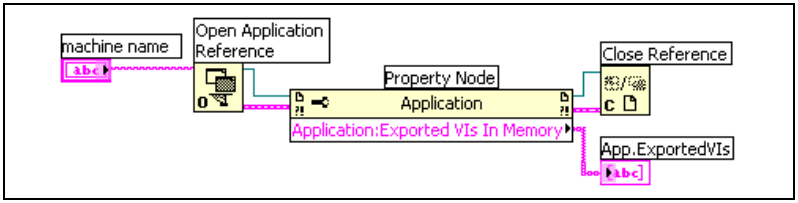


Figure 17-2. Displaying All VIs in Memory on a Remote Computer

Manipulating VI Class Properties and Methods

You can get or set properties of a VI, perform methods on a VI, or both. In Figure 17-3, LabVIEW reinitializes the front panel objects of a VI to their default values using the Invoke Node. The front panel opens and displays the default values using the Property Node.

If you do not wire a refnum to the **reference** input, the Property Node or Invoke Node uses a reference to the VI containing the Property Node or Invoke Node. If you want to manipulate the properties or methods of another VI, you must wire a VI refnum to the **reference** input.

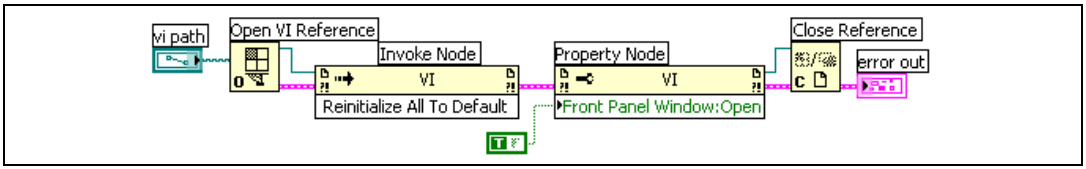


Figure 17-3. Using VI Class Property and Invoke Nodes

The Property Node operates similarly to the Invoke Node. After you wire a VI refnum to the Property Node, you can access all the VI Class properties.

Manipulating Application and VI Class Properties and Methods

In some VIs, you must access both Application and VI Class properties or methods. You must open and close the Application and VI Class refnums separately, as shown in Figure 17-4.

Figure 17-4 shows how to determine the VIs in memory on a local computer and to display the path to each of the VIs on the front panel. To find all the VIs in memory, you must access an Application Class property. To determine the paths to each of these VIs, you must access a VI Class property. The number of VIs in memory determines the number of times the For Loop executes. Place the Open VI Reference and Close Reference functions inside the For Loop because you need a VI refnum for each VI in memory. Do not close the Application refnum until the For Loop finishes retrieving all the VI paths.

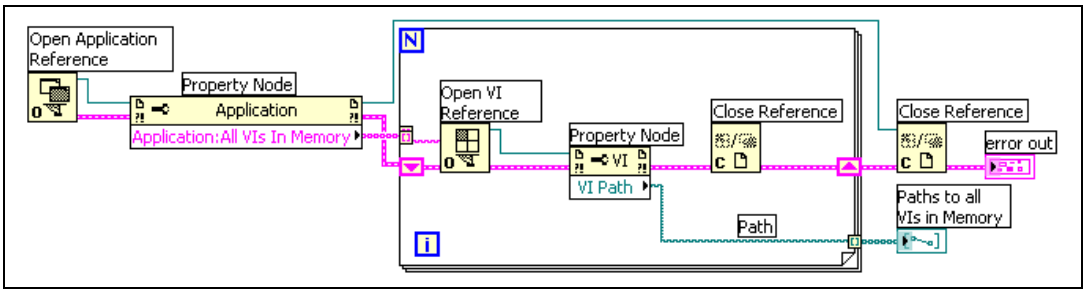


Figure 17-4. Using Application and VI Class Properties and Methods

Dynamically Loading and Calling VIs

You can dynamically load VIs instead of using statically linked subVI calls. A statically linked subVI is one you place directly on the block diagram of a caller VI. It loads at the same time the caller VI loads.

Unlike statically linked subVIs, dynamically loaded subVIs do not load until the caller VI makes the call to the subVI. If you have a large caller VI, you can save load time and memory by dynamically loading the subVI because the subVI does not load until the caller VI needs it, and you can release it from memory after the operation completes.

Call By Reference Nodes and Strictly Typed VI Refnums

Use the Call By Reference Node to dynamically call VIs.

The Call By Reference Node requires a strictly typed VI refnum. The strictly typed VI refnum identifies the connector pane of the VI you are calling. It does not create a permanent association to the VI or contain other VI information, such as the name and location. You can wire the Call By Reference Node inputs and outputs just like you wire any other VI.

Figure 17-5 shows how to use the Call By Reference Node to dynamically call the Frequency Response VI. The Call By Reference Node requires the use of the Open VI Reference and the Close Reference functions, similar to the functions you use for the Property Node and Invoke Node.

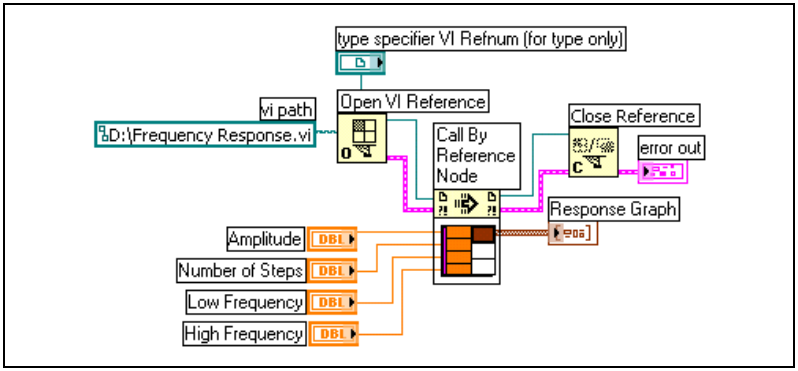


Figure 17-5. Using the Call By Reference Node

The VI you specify for strictly typed refnums provides only the connector pane information. That is, no permanent association is made between the refnum and the VI. In particular, avoid confusing selecting the VI connector pane with getting a refnum to the selected VI. You specify a particular VI using the **vi path** input on the Open VI Reference function.

Editing and Running VIs on Remote Computers

An important aspect of both Application and VI refnums is their network transparency. This means you can open refnums to objects on remote computers in the same way you open refnums to those objects on your computer.

After you open a refnum to a remote object, you can treat it in exactly the same way as a local object, with a few restrictions. For operations on a remote object, the VI Server sends the information about the operation across the network and sends the results back. The application looks almost identical regardless of whether the operation is remote or local.

Controlling Front Panel Objects

Control references correspond to user interface object references in text-based programming languages. Control references do not correspond to pointers in text-based programming languages.

Use the control refnum controls located on the **Refnum** and **Classic Refnum** palettes to pass front panel object references to other VIs. You also can right-click a front panel object and select **Create»Reference** from the shortcut menu to create a control reference. After you pass a control

reference to a subVI, use Property Nodes and Invoke Nodes to read and write properties and invoke methods of the referenced front panel object.

Refer to the *Case and Sequence Structures* section of Chapter 8, *Loops and Structures*, for information about using events to control block diagram behavior programmatically through front panel objects.

Strictly Typed and Weakly Typed Control Refnums

Strictly typed control refnums accept only control refnums of exactly the same data type. For example, if the type of a strictly typed control refnum is 32-bit integer slide, you can wire only a 32-bit integer slide to the control refnum terminal. You cannot wire an 8-bit integer slide, a double-precision scalar slide, or a cluster of 32-bit integer slides to the control refnum terminal.

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

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

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

Note When you wire a Property Node to a weakly typed control reference terminal, the Value property produces variant data, which might require conversion before you can use the data. The History Data property for charts is available only if the chart reference is strictly typed. Refer to the *Handling Variant Data* section of Chapter 5, *Building the Block Diagram*, for more information about variant data.

18

Networking in LabVIEW

VIs can communicate, or network, with other processes, including those that run on other applications or on remote computers. Use the networking features in LabVIEW to perform the following tasks:

- Share live data with other VIs running on a network using National Instruments DataSocket technology.
- Publish front panel images and VI documentation on the Web.
- Email data from VIs.
- Build VIs that communicate with other applications and VIs through low-level protocols, such as TCP, UDP, Apple events, and PPC Toolbox.

For more information…

Refer to the *LabVIEW Help* for more information about networking in LabVIEW.

Choosing among File I/O, VI Server, ActiveX, and Networking

Networking might not be the best solution for your application. If you want to create a file that contains data other VIs and applications can read, use the File I/O VIs and functions. Refer to Chapter 14, *File I/O*, for more information about using the File I/O VIs and functions.

If you want to control other VIs, use the VI Server. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about controlling VIs and other LabVIEW applications on local and remote computers.

(Windows) If you want to access the features of many Microsoft applications, such as embedding a waveform graph in an Excel spreadsheet, use the ActiveX VIs and functions. Refer to Chapter 19, *Windows Connectivity*, for more information about accessing ActiveX-enabled applications and permitting other ActiveX applications access to LabVIEW.

LabVIEW as a Network Client and Server

You can use LabVIEW as a client to subscribe to data and use features in other applications or as a server to make LabVIEW features available to other applications. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using the VI Server to control VIs on local and remote computers. You control VIs by accessing properties and invoking methods using the Property Node and Invoke Node, respectively.

Before you can access the properties and invoke methods of another application, you must establish the network protocol you use to access the properties and methods. Protocols you can use include HTTP and TCP/IP. The protocol you select depends on the application. For example, the HTTP protocol is ideal for publishing on the Web, but you cannot use the HTTP protocol to build a VI that listens for data that another VI creates. To do that, use the TCP protocol.

Refer to the *Low-Level Communications Applications* section of this chapter for more information about communications protocols LabVIEW supports.

(Windows) Refer to Chapter 19, *Windows Connectivity*, for more information about using ActiveX technology with LabVIEW as an ActiveX server or client.

Using DataSocket Technology

Use National Instruments DataSocket technology to share live data with other VIs on the Web or on your local computer. DataSocket pulls together established communication protocols for measurement and automation in much the same way a Web browser pulls together different Internet technologies.

DataSocket technology provides access to several input and output mechanisms from the front panel through the **DataSocket Connection** dialog box. Right-click a front panel object and select **Data Operations»** **DataSocket Connection** from the shortcut menu to display the **DataSocket Connection** dialog box. You publish (write) or subscribe (read) to data by specifying a URL, in much the same way you specify URLs in a Web browser.

For example, if you want to share the data in a thermometer indicator on the front panel with other computers on the Web, publish the thermometer data by specifying a URL in the **DataSocket Connection** dialog box. Users on other computers subscribe to the data by placing a thermometer on their front panel and selecting the URL in the **DataSocket Connection** dialog box. Refer to the *Using DataSocket on the Front Panel* section of this chapter for more information about using DataSocket technology on the front panel.

Refer to the *Integrating the Internet into Your Measurement System* white paper for more information about DataSocket technology. This white paper is available as a PDF from the Installation CD, in the `manuals` directory, or from the National Instruments Web site at `ni.com`.

Specifying a URL

URLs use communication protocols, such as `dstp`, `ftp`, and `file`, to transfer data. The protocol you use in a URL depends on the type of data you want to publish and how you configure your network.

You can use the following protocols when you publish or subscribe to data using DataSocket:

- **DataSocket Transport Protocol** (`dstp`)—The native protocol for DataSocket connections. When you use this protocol, the VI communicates with the DataSocket Server. You must provide a named tag for the data, which is appended to the URL. The DataSocket connection uses the named tag to address a particular data item on a DataSocket Server. To use this protocol, you must run a DataSocket Server.
- **(Windows) OLE for Process Control** (`opc`)—Designed specifically for sharing real-time production data, such as data generated by industrial automation operations. To use this protocol, you must run an OPC server.
- **(Windows)** `logos`—An internal National Instruments technology for transmitting data between the network and your local computer.
- **File Transfer Protocol** (`ftp`)—You can use this protocol to specify a file from which to read data.

Note To read a text file from an FTP site using DataSocket, add `[text]` to the end of the DataSocket URL.

- `file`—You can use this protocol to provide a link to a local or network file that contains data.

Table 18-1 shows examples of each protocol URL.

Table 18-1. Example DataSocket URLs

URL	Example
dstp	dstp://servername.com/numeric, where numeric is the named tag for the data
opc	opc:\National Instruments.OPCTest\item1 opc:\\machine\National Instruments.OPCModbus\Modbus Demo Box.4:0 opc:\\machine\National Instruments.OPCModbus\Modbus Demo Box.4:0?updaterate=100&deadband=0.7
logos	logos://computer_name/process/data_item_name
ftp	ftp://ftp.ni.com/datasocket/ping.wav
file	file:ping.wav file:c:\mydata\ping.wav file:\\machine\mydata\ping.wav

Use the dstp, opc, and logos URLs to share live data because these protocols can update remote and local controls and indicators. Use the ftp and file URLs to read data from files because these protocols cannot update remote and local controls and indicators.

Refer to the examples\comm\datasktx.llb for examples of using DataSocket connections.

Data Formats Supported by DataSocket

Use DataSocket to publish and subscribe to the following data:

- **Raw text**—Use raw text to deliver a string to a string indicator.
- **Tabbed text**—Use tabbed text, as in a spreadsheet, to publish data in arrays. LabVIEW interprets tabbed text as an array of data.
- .wav **data**—Use .wav data to publish a sound to a VI or function.
- **Variant data**—Use variant data to subscribe to data from another application, such as a National Instruments Measurement Studio ActiveX control.

Using DataSocket on the Front Panel

Use front panel DataSocket connections to publish or subscribe to live data in a front panel object. When you share the data of a front panel object with other users, you publish data. When users retrieve the published data and view it on their front panel, users subscribe to the data.

DataSocket connections differ from Web Server and ActiveX connections because you can use DataSocket connections directly from the front panel without any block diagram programming. Each front panel control or indicator can publish or subscribe to data through its own DataSocket connection. Front panel DataSocket connections publish only the data, not a graphic of the front panel control, so the VIs that subscribe through a DataSocket connection can perform their own operations on the data.

You can set the value of a front panel control directly on the front panel and then publish the data, or you can build a block diagram, wire the output of a VI or function to an indicator, and publish the data from that indicator. Typical scenarios for using DataSocket connections with controls and indicators include the following:

- Publish a value from a front panel control to manipulate a control and publish the data for other users to subscribe to through a control or indicator. For example, if you place a knob on your front panel that raises or lowers temperature, a user on another computer can subscribe to the data and use it in a control wired to a subVI or function, or view the data in an indicator.
- Publish a value that appears in a front panel indicator so another user can subscribe to the data and view the data in an control or indicator on the front panel, or use the results as data in a control wired to an input in a subVI or function. For example, a VI that calculates the mean temperature and displays the temperature in a thermometer on the front panel can publish the temperature data.
- Subscribe to a value that appears in a control or indicator on the front panel of another VI to view data in a front panel indicator on your VI. If you subscribe to the data with a control, you can use the data in your VI by wiring the control to an input of a subVI or function.
- Publish from and subscribe to a front panel control so users can manipulate a control on the front panel of your VI from the front panels of their VIs. When you run the VI, the front panel control on your VI retrieves the current value that another VI or application published through the DataSocket connection. When a user changes the control value on the front panel, the DataSocket connection publishes the new value to the front panel control of your VI. If you then manipulate the

value of your front panel control, your VI publishes the value to the front panels of other users.

The front panel objects that subscribe to data do not have to be the same kind of objects that publish the data. However, the front panel objects must be the same data type or, if they are numeric data, they must coerce. For example, you can use a digital indicator in your VI to view the data a thermometer in another VI generates. The thermometer can be a floating-point number, and the digital indicator can be an integer.

Front panel DataSocket connections are primarily intended for sharing live data. To read data in local files, FTP servers, or Web servers, use the DataSocket Read function, the File I/O VIs and functions, or the Application Control VIs and functions.

Reading and Writing Live Data through the Block Diagram

From the block diagram, you can programmatically read or write data using the DataSocket functions.

Use the DataSocket Write function to write live data through a DataSocket connection programmatically. Figure 18-1 shows how to write a numeric value.

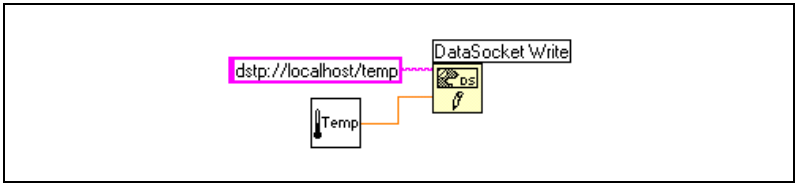


Figure 18-1. Publishing Data Using DataSocket Write

The DataSocket Write function is polymorphic, so you can wire most data types to the **data** input. Refer to the *Polymorphic VIs and Functions* section of Chapter 5, *Building the Block Diagram*, for more information about polymorphic VIs and functions.

Use the DataSocket Read function to read live data from a DataSocket connection programmatically. Figure 18-2 shows how to read data and convert it to a double-precision floating-point number.

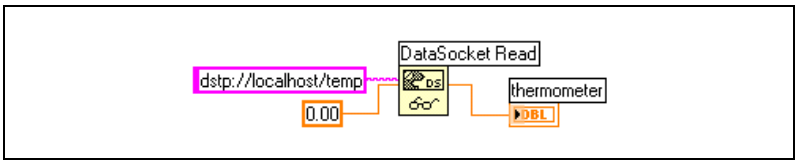


Figure 18-2. Reading a Single Value Using DataSocket Read

Convert live data to a specific data type by wiring a control or constant to the **type** input of DataSocket Read. If you do not specify a type, the **data** output of DataSocket Read returns variant data, which you must manipulate with the Variant to Data function.

Programmatically Opening and Closing DataSocket Connections

Use the DataSocket Open and DataSocket Close functions to control when a DataSocket connection opens and closes. When you open a DataSocket connection using the DataSocket Open function, the connection remains open until one of the following conditions are met: you explicitly close the connection using the DataSocket Close function, you close the VI, or the VI stops running. The **URL** input of the DataSocket Open function accepts only a DataSocket URL. The DataSocket Open function returns a DataSocket connection refnum that you can use as the **URL** input for the DataSocket Read and DataSocket Write functions.

Buffering DataSocket Data

If you use the DataSocket Transport Protocol (`dstp`), the DataSocket Server by default publishes only the most recent value to all subscribers. When one client publishes values to the server faster than another client reads them, newer values overwrite older, unprocessed values before the clients read them. This loss of unprocessed data can occur at the server or at the client. This loss of data might not be a problem if you are subscribing to DataSocket data and you want to receive only the most recent value published to the server. However, if you want to receive every value published to the server, you must buffer the data on the client.

Note Client-side buffering also applies to other protocols, such as `opc`, `logos`, and `file`. To use `dstp` buffering, you also must use the DataSocket Server Manager to configure server-side buffering. Refer to the *DataSocket Help* for more information about server-side buffering.

`dstp` buffering does not guarantee data delivery. If the data in a buffer at the server or client exceeds the buffer size, the buffer discards older values in place of newer values. To detect discarded values in a data stream, wire the published data to the Set Variant Attribute function to uniquely identify each value in the publisher and check for discarded sequence IDs in the subscriber.

Set the **mode** input of the DataSocket Open function to **BufferedRead** or **BufferedReadWrite** and use a Property Node to set the DataSocket properties for the size of a FIFO buffer. Doing so ensures that LabVIEW stores the values the client receives in a buffer rather than overwriting them every time the value changes.

Note If you use DataSocket properties to set the size of a FIFO buffer, you must set the **mode** input of the DataSocket Open function to **BufferedRead** or **BufferedReadWrite**. Otherwise, the item at the server is not buffered for the connection.

Figure 18-3 uses DataSocket buffering.

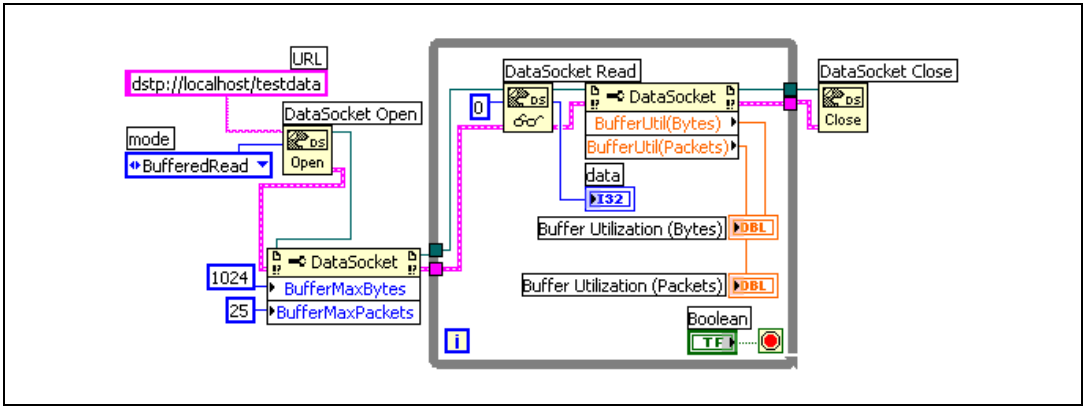


Figure 18-3. DataSocket Buffering

Note Buffering applies only when you use the DataSocket Read function to subscribe to data a server publishes. Buffering is not available when you use front panel DataSocket connections to subscribe to data.

Reporting Diagnostics

Use the Buffer Utilization (Bytes) property or the Buffer Utilization (Packets) property to request diagnostic information about the buffers you specify. Use these properties to check the percentage of buffer in use on the client to determine whether the current buffer size is sufficient. If the value

of either of these properties approaches the maximum value of the buffer, increase the buffer size to make sure you receive all values the server publishes.

Refer to the *LabVIEW Help* for more information about specifying the buffer size for a DataSocket client.

DataSocket and Variant Data

In some cases, the VI or other application that programmatically reads the data cannot convert the data back into its original data type, such as when you subscribe to data from another application. Also, you might want to add an attribute to the data, such as a time stamp or warning, which the data types do not permit.

In these cases, use the To Variant function to programmatically convert the data you write to a DataSocket connection to variant data. Figure 18-4 shows a block diagram that continually acquires a temperature reading, converts the data to variant data, and adds a time stamp as an attribute to the data.

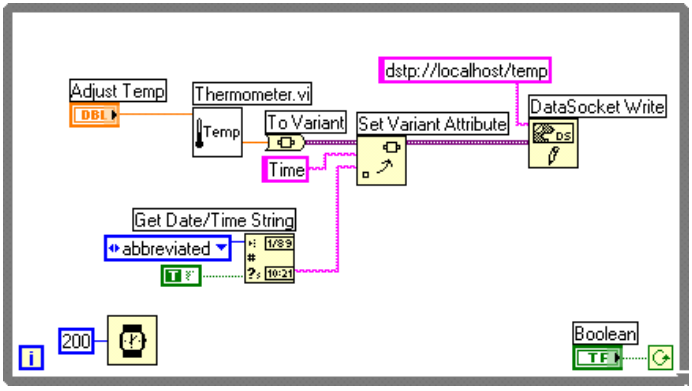


Figure 18-4. Converting Live Temperature Data to Variant Data

When another VI reads the live data, the VI must convert the variant data to a data type it can manipulate. Figure 18-5 shows a block diagram that continually reads temperature data from a DataSocket connection, converts the variant data into a temperature reading, retrieves the time stamp attribute associated with each reading, and displays the temperature and the time stamp on the front panel.

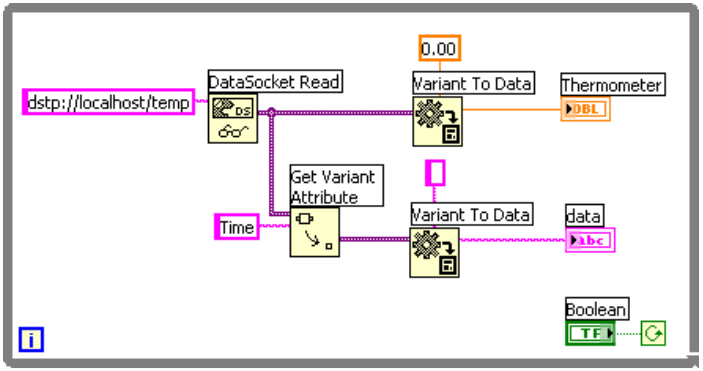


Figure 18-5. Converting Live Variant Data to Temperature Data

Publishing VIs on the Web

Use the LabVIEW Web Server to create HTML documents, publish front panel images on the Web, and embed VIs in a Web page. You can control browser access to the published front panels and configure which VIs are visible on the Web.

Note Use the LabVIEW Enterprise Connectivity Toolset to control VIs on the Web and to add more security features to VIs you publish on the Web. Refer to the National Instruments Web site at `ni.com` for more information about this toolset.

Web Server Options

Select **Tools»Options** and select one of the **Web Server** items in the top pull-down menu to set the following options:

- Set up the root directory and log file.
- Enable the Web Server.
- Control browser access to VI front panels.
- Configure which VI front panels are visible on the Web.

You must enable the Web Server in the **Web Server: Configuration** page of the **Options** dialog box before you can publish VIs on the Web. You also can enable the Web Server with the Web Publishing Tool, described in the *Creating HTML Documents* section of this chapter. The VIs must be in memory before you publish them.

Creating HTML Documents

Select **Tools»Web Publishing Tool** to use the Web Publishing Tool to accomplish the following tasks:

- Create an HTML document.
- Embed static or animated images of the front panel in an HTML document. Currently only Netscape browsers support animated images.
- Embed a VI that clients can view and control remotely.
- Add text above and below the embedded VI front panel image.
- Place a border around an image or embedded VI.
- Preview the document.
- Save the document to disk.
- Enable the Web Server for publishing HTML documents and front panel images on the Web.

Publishing Front Panel Images

Use a `.snap` URL in a Web browser or an HTML document to return a static image of the front panel of a VI currently in memory. The query parameters in the URL specify the VI name and the attributes of the image. For example, use `http://`*`web.server.address`*`/.snap?`*`VIName.vi`*, where *`VIName.vi`* is the name of the VI you want to view.

Use a `.monitor` URL to return an animated image of the front panel of a VI currently in memory. The query parameters in the URL specify the VI name, attributes of the animation, and attributes of the image. For example, use `http://`*`web.server.address`*`/.monitor?`*`VIName.vi`*, where *`VIName.vi`* is the name of the VI you want to view.

Front Panel Image Formats

The Web Server can generate images of front panels in the JPEG and PNG image formats.

The JPEG format compresses graphics well but can lose some graphic detail. This format works best for photos. For line art, front panels, and block diagrams, JPEG compression can result in fuzzy graphics and uneven colors. The PNG format also compresses graphics well, although not always as well as the JPEG format. However, PNG compression does not lose any detail.

Viewing and Controlling Front Panels Remotely

You can view and control a VI front panel remotely, either from within LabVIEW or from within a Web browser, by connecting to the LabVIEW built-in Web Server. When you open a front panel remotely from a client, the Web Server sends the front panel to the client, but the block diagram and all the subVIs remain on the server computer. You can interact with the front panel in the same way as if the VI were running on the client, except the block diagram executes on the server. Use this feature to publish entire front panels or to control your remote applications safely, easily, and quickly.

Note Use the LabVIEW Web Server if you want to control entire VIs. Use the DataSocket Server to read and write data on a single front panel control in a VI. Refer to the *Using DataSocket Technology* section of this chapter for more information about using the DataSocket Server.

Configuring the Server for Clients

The user at the server computer must first configure the server before a client can view and control a front panel remotely using LabVIEW or a Web browser. Configure the Web Server by selecting **Tools»Options** and selecting the **Web Server** pages from the top pull-down menu. Use these pages to control browser access to the server and to specify which front panels are visible remotely. You also can use these pages to set a time limit on how long a remote client can control a VI when multiple clients are waiting to control the VI.

The Web Server allows multiple clients to connect simultaneously to the same front panel, but only one client at a time can control the front panel. The user at the server computer can regain control of any VI at any time. When the controller changes a value on the front panel, all client front panels reflect that change. However, client front panels do not reflect all changes. In general, client front panels do not reflect changes made to the display on front panel objects, but rather to the actual values in the front panel objects. For example, if the controller changes the mapping mode or marker spacing of a scale of a chart or if the controller shows and hides a scrollbar for a chart, only the controller front panel reflects these changes.

Remote Panel License

You must configure a license to support the number of clients that potentially could connect to your server.

Note **(Windows 9.*x*)** The LabVIEW Full Development System and the Application Builder include a remote panel license that allows one client to view and control a front panel remotely. The LabVIEW Professional Development System includes a remote panel license that allows five clients to view and control a front panel remotely. You cannot upgrade the remote panel license to support more clients on Windows 9*x*.
(Windows 2000/NT/XP, Mac OS, and UNIX) The LabVIEW Full Development System and the Application Builder include a remote panel license that allows one client to view and control a front panel remotely. The LabVIEW Professional Development System includes a remote panel license that allows five clients to view and control a front panel remotely. You can upgrade the remote panel license to support more clients.

Viewing and Controlling Front Panels in LabVIEW or from a Web Browser

A client can view and control a front panel remotely from LabVIEW or from a Web browser.

Note Before you can view and control a front panel remotely, you must enable the Web Server on the server computer where the VI you want to view and control is located.

Viewing and Controlling Front Panels in LabVIEW

To view a remote front panel using LabVIEW as a client, open a new VI and select **Operate»Connect to Remote Panel** to display the **Connect to Remote Panel** dialog box. Use this dialog box to specify the server Internet address and the VI you want to view. By default, the remote VI front panel is initially in observer mode. You can request control by placing a checkmark in the **Request Control** checkbox in the **Connect to Remote Panel** dialog box when you request a VI. When the VI appears on your computer, you also can right-click anywhere on the front panel and select **Request Control** from the shortcut menu. You also can access this menu by clicking the status bar at the bottom of the front panel window. If no other client is currently in control, a message appears indicating that you have control of the front panel. If another client is currently controlling the VI, the server queues your request until the other client relinquishes control or until the control time limit times out. Only the user at the server computer can monitor the client queue list by selecting **Tools»Remote Panel Connection Manager**.

If you want to save the data generated by a VI running on a remote computer, use DataSocket or TCP instead of remote front panels. Refer to the *Using DataSocket Technology* section of this chapter for more information about using DataSocket. Refer to the *TCP and UDP* section of this chapter for more information about using TCP.

All VIs you want clients to view and control must be in memory. If the requested VI is in memory, the server sends the front panel of the VI to the requesting client. If the VI is not in memory, the **Connection Status** section of the **Open Remote Panel** dialog box displays an error message.

Viewing and Controlling Front Panels from a Web Browser

If you want clients who do not have LabVIEW installed to be able to view and control a front panel remotely using a Web browser, they must install the LabVIEW Run-Time Engine. The LabVIEW Run-Time Engine includes a LabVIEW browser plug-in package that installs in the browser plug-in directory. The LabVIEW CD contains an installer for the LabVIEW Run-Time Engine.

Clients install the LabVIEW Run-Time Engine and the user at the server computer creates an HTML file that includes an `<OBJECT>` and `<EMBED>` tag that references the VI you want clients to view and control. This tag contains a URL reference to a VI and information that directs the Web browser to pass the VI to the LabVIEW browser plug-in. Clients navigate to the Web Server by entering the Web address of the Web Server in the address or URL field at the top of the Web browser window. The plug-in displays the front panel in the Web browser window and communicates with the Web Server so the client can interact with the remote front panel. Clients request control by selecting **Request Control of VI** at the bottom of the remote front panel window in their Web browser or by right-clicking anywhere on the front panel and selecting **Request Control of VI** from the shortcut menu.

If no other client is currently in control and if no other browser window on the same connection is currently in control, a message appears indicating that you have control of the front panel. If another client is currently controlling the VI, the server queues your request until the other client relinquishes control or until the control time limit times out. Only the user at the server computer can monitor the client queue list by selecting **Tools»Remote Panel Connection Manager**.

Note National Instruments recommends that you use Netscape 4.7 or later or Internet Explorer 5.5 Service Pack 2 or later when viewing and controlling front panels in a Web browser.

Functionality Not Supported in Viewing and Controlling Remote Front Panels

Because of the constraints of a Web browser, user interface applications that attempt to manipulate the dimensions and location of a front panel do not work properly when that front panel is displayed as a part of a Web page. Although the Web Server and the LabVIEW browser plug-in attempt to preserve the fidelity of complex user interface applications—in particular, those that present dialog boxes and subVI windows—some applications might not work properly in the context of a Web browser. National Instruments recommends that you do not export these types of applications for use in a Web browser.

Avoid exporting VIs that have While Loops but no wait function. These VIs prevent background tasks from performing in a reasonable amount of time, making front panels unresponsive when viewed or controlled remotely.

Additionally, some VIs might not work exactly the same way from a remote computer as they do when run locally. ActiveX controls embedded on a front panel do not display on a remote client because they draw and operate almost completely independent of LabVIEW. If a VI presents the standard file dialog box, the controller receives an error because you cannot browse a file system remotely. Also, the browse button of a path control is disabled in remote panels.

Clients viewing a front panel remotely might see different behavior depending on whether the front panel they are connecting to is from a built application. Specifically, if the front panel is from a built application, any programmatic changes to the front panel made before the client connects to the front panel are not reflected on the client computer. For example, if a Property Node changes a caption on a control before a client connects to that front panel, the client will see the original caption of the control, not the changed caption.

Only a controller can remotely view the front panel of a VI dynamically opened and run using the VI Server or the front panel of a subVI configured to display the front panel when called. Clients not controlling the VI cannot view the front panel.

Block diagrams that achieve certain user interface effects by polling properties of a front panel control might experience decreases in performance when you control the VI from a remote computer. You can improve the performance of these VIs by using the Wait for Front Panel Activity function.

Emailing Data from VIs

Use the SMTP E-mail VIs to send electronic mail, including attached data and files, using the Simple Mail Transfer Protocol (SMTP). LabVIEW does not support authentication for SMTP. You cannot use the SMTP E-mail VIs to receive information. The SMTP E-mail VIs use the Multipurpose Internet Mail Extensions (MIME) format to encode the message. In this format, you can send multiple documents, including binary data files, within an email. You also can describe properties of the individual attachments, such as which character set a text document uses. Refer to the *LabVIEW Help* for information about emailing data from a VI.

Note The SMTP E-mail VIs are available only in the Full and Professional Development Systems.

In addition to the recipient(s) email address, you must know the Web address of the SMTP server. You specify a mail server by wiring a **mail server** to every SMTP E-mail VI you use. The **mail server** must be the host name or IP address of an external server computer that can service requests from the computer running the SMTP E-mail VIs. If you do not know which mail server to use, contact your network administrator for the name of a valid server. After you specify a valid mail server, the SMTP E-mail VIs open a connection to the server and send the server commands that describe the recipients and the contents of the email. The server sends the message to the individual recipients or forwards it to other SMTP servers. Refer to the `examples\comm\smtpex.llb` for examples of using the SMTP E-mail VIs.

Note The SMTP E-mail VIs do not support multi-byte characters, such as Japanese.

Selecting a Character Set

The **character set** input parameter of the SMTP E-mail VIs specifies the character set to use in the text of the email or attachment. A character set describes the mapping between characters and their character codes.

A character is a basic unit of written languages—a letter, a number, a punctuation mark, or, in some languages, an entire word. Modifications of letters, such as capitalization or accent marks, make that letter a separate character. For example, `O`, `o`, Ö, and Ô are separate characters.

A character code is a number that represents a character. Because computers interpret only numbers, they must associate a character with a number to operate on the character.

A character set is the relationship between characters and the numbers that represent them in the computer. For example, in the ASCII character set, the character codes for `A`, `B`, and `C`, are `65`, `66`, and `67`, respectively.

US-ASCII Character Set

A commonly used character set for email is the US-ASCII, or ASCII. Many email applications use this character set as the default and do not work with other sets. The ASCII character set describes the letters and most punctuation marks used in the English language, for a total of 128 characters. Most other character sets are extensions of ASCII.

Using the ASCII character set might not be sufficient because many languages require characters not found in ASCII. For example, you cannot write the German word `Müller` using ASCII because the character `ü` is not defined in the ASCII character set.

ISO Latin-1 Character Set

Because many languages require characters not found in ASCII, countries that use these languages created new character sets. Most of these character sets contain the first 128 character codes as ASCII and define the next 128 character codes to describe characters needed in that language. Some of these character sets use different character codes to describe the same characters. This can cause problems when one character set displays text written in another character set. To solve this problem, use a standard character set. One widely used standard character set is ISO Latin-1, also known as ISO-8859-1. This character set describes characters used in most western European languages and most email applications that handle those languages include the set.

Mac OS Character Set

Apple Computer, Inc. developed its own extended character set before ISO Latin-1 was defined. The Mac OS character set is based on ASCII but uses a different set of upper 128 character codes than ISO Latin-1. Because of this, email that contains accented characters written in the Mac OS character set appears incorrectly in email applications that expect ISO Latin-1 text. To solve this problem, Mac OS email applications convert text to ISO Latin-1 before mailing it. When another Mac OS email application

receives text designated as using the ISO Latin-1 character set, it converts the text to the Mac OS character set.

Transliteration

Transliteration is the mapping of characters to another character set. Use transliteration if you are sending an email and you need to change the text of the email to the corresponding characters in another character set. You can use the SMTP E-mail VIs to specify character sets that map text to another character set before sending the text. For example, if you create a message using standard ASCII characters and specify that the character set is MacRoman, the SMTP E-mail VIs transliterate the text and send it as character set `iso-8859-1` (ISO Latin-1). Use the **translit** input parameter of the SMTP E-mail VIs to specify which transliterations the VIs can use. A transliteration is defined by a virtual character set, a target character set, and a transliteration, or mapping, file. The transliteration file specifies that one character maps to another character.

The transliteration file is a binary file of 256 bytes with 256 entries. Each entry in the file corresponds to a character in the virtual character set, and contains the new character code from the target character set. For example, if you want to map the `a` character, whose code is `61`, to the `A` character, whose code is `41`, the entry with an index of `61` in the transliteration file should contain the value `41`. If an entry contains a value that equals its index, the transliteration file does not modify the character in the virtual character set. For example, if the entry with an index of `61` in the transliteration file contains a value of `61`, the transliteration file does not modify the character.

When you specify a transliteration file as the target character set in the **translit** input, the mappings are applied in the order specified. For example, if the transliteration entry is [MacRoman iso-8859-1 `macroman.trl`, MacRomanUp MacRoman `asciiup.trl`], the character set MacRoman changed to iso-8859-1 using `macroman.trl` and then MacRomanUp changed to MacRoman using `asciiup.trl`. Refer to the `vi.lib\Utility\SMTP` directory for examples of `.trl` files.

Low-Level Communications Applications

LabVIEW supports several low-level protocols you can use to communicate between computers.

Each protocol is different, especially in the way it refers to the network location of a remote application. Each protocol generally is incompatible with other protocols. For example, if you want to communicate between Mac OS and Windows, you must use a protocol that works on both platforms, such as TCP.

TCP and UDP

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are available on all platforms LabVIEW supports. TCP is a reliable, connection-based protocol. It provides error detection and ensures that data arrive in order and without duplication. For these reasons, TCP is usually the best choice for network applications.

Although UDP can give higher performance than TCP and does not require a connection, it does not guarantee delivery. Typically, use UDP in applications in which guaranteed delivery is not critical. For example, an application might transmit data to a destination frequently enough that a few lost segments of data are not problematic. Refer to the *Using LabVIEW with TCP/IP and UDP* Application Note for more information about using TCP and UDP in LabVIEW applications.

Use the UDP Multicast Open VI instead of the UDP Open function to open connections capable of reading, writing, or reading and writing UDP data to or from a multicase IP address. A multicast IP address defines a multicast group. Multicast IP addresses are in the `224.0.0.0` to `239.255.255.255` range. When a client wants to join a multicast group, it subscribes to the multicast IP address of the group. After the client subscribes to a multicast group, the client receives data sent to the multicast IP address. Refer to the *Using LabVIEW with TCP/IP and UDP* Application Note for more information about using UDP multicasting.

Apple Events and PPC Toolbox (Mac OS)

The more common Mac-only form of communication is Apple events. Use Apple events to send messages to request actions or return information from other Mac OS applications.

Program-to-Program Communication (PPC) Toolbox is a lower level form of sending and receiving data between Mac OS applications. PPC Toolbox gives higher performance than Apple events because the overhead required to transmit information is lower. However, because PPC Toolbox does not define the types of information you can transfer, many applications do not support it. PPC Toolbox is the best method to send large amounts of information between applications that support PPC Toolbox. Refer to the *Using Apple Events and the PPC Toolbox to Communicate with LabVIEW Applications on the Macintosh* Application Note for more information about using Apple events and PPC Toolbox in LabVIEW applications.

Pipe VIs (UNIX)

Use the Pipe VIs to open, close, read, and write to UNIX named pipes. Use named pipes to communicate between LabVIEW and unrelated processes.

Executing System-Level Commands (Windows and UNIX)

Use the System Exec VI to execute or launch other Windows applications or UNIX command-line applications from within VIs. With the System Exec VI, you execute a system-level command line that can include any parameters supported by the application you want to launch.

19

Windows Connectivity

LabVIEW provides access to other Windows applications using .NET or ActiveX technologies. You can use LabVIEW as a .NET client to access the objects, properties, and methods associated with .NET servers. LabVIEW is not a .NET server. Other applications cannot directly communicate with LabVIEW through .NET. With a .NET-enabled VI, you can connect to Windows services and APIs. The .NET framework includes COM+ component services, ASP Web development framework, and support for Web service protocols, such as, SOAP, WSDL, and UDDI. The .NET framework is the programming basis of the .NET environment you use to build, deploy, and run Web-based applications, smart client applications, and XML Web services.

With ActiveX Automation, a Windows application, such as LabVIEW, provides a public set of objects, commands, and functions that other Windows applications can access. You can use LabVIEW as an ActiveX client to access the objects, properties, methods, and events associated with other ActiveX-enabled applications. LabVIEW also can act as an ActiveX server, so other applications can access LabVIEW objects, properties, and methods.

.NET refers to Microsoft's .NET technology. You must install the .NET framework. Refer to the MSDN Web site for more information about .NET and installing the framework. *ActiveX* refers to Microsoft's ActiveX technology and OLE technology. Refer to the Microsoft Developer's Network documentation, *Inside OLE*, by Kraig Brockschmidt, second edition, and *Essential COM*, by Don Box, for more information about ActiveX.

For more information…

Refer to the *LabVIEW Help* and the National Instruments Web site at `ni.com` for more information about using .NET and ActiveX technology.

.NET Environment

The following list provides a background of the various elements that make up the .NET environment. The purpose of this information is to help you understand .NET, but learning this information is not essential for you to use the .NET components in LabVIEW.

- **Common Language Runtime (CLR)**—a set of libraries responsible for run-time services, such as language integration, security enforcement, memory, garbage collection, process management, and thread management. The CLR forms the foundation of .NET and uses the intermediate language (IL) that all programming languages generate to facilitate communication between .NET and other programs.

  To help .NET communicate with various programs, the CLR provides a data type system that spans programming languages and operating system boundaries. Developers use CLR to view the system as a collection of data types and objects rather than as a collection of memory and threads. The CLR requires compilers and linkers to generate information in the CLR IL metadata format. In a Win32 system, all programming language compilers generate CLR IL code rather than the assembly code.
- **Class Libraries**—a set of classes that provides standard functionality, such as input and output, string manipulation, security management, network communications, thread management, text management, user interface design features, and so on. These classes contain the same functions the Win32/COM system uses. In the .NET framework, you can use classes created in one .NET language in another .NET language.
- **Assemblies**—a unit of deployment similar to a DLL, OCX, or executable for a component in COM. Assemblies are DLLs and executables you build using the .NET CLR. Assemblies can consist of a single file or multiple files. An assembly includes a manifest that contains information about the assembly name, version information, local information, publisher's security information, list of files that make up the assembly, list of dependent assemblies, resources, and exported data types. Single-file assemblies contain all the data in a single file, including the manifest and any resources it needs. Multi-file assemblies might have external resources, such as bitmaps, icons, sound files, and so on, or have one file for the core code and another for helper libraries.

Assemblies can be public or private. .NET requires that private assemblies be in the same directory as the application directory and that public assemblies be in a system wide global cache called the Global Assembly Cache (GAC). The developer of the application typically writes private assemblies for use by that application. The developer of the assembly also decides the version control. The assembly name is the filename, minus any file extension, of the file that contains the manifest.

- **Global Assembly Cache (GAC)**—a listing of the public assemblies available on the system. The GAC is similar to the registry COM uses.

.NET Functions and Nodes

Use the following LabVIEW functions and nodes to access the objects, properties, and methods associated with .NET servers.

- Use the Constructor Node located on the **.NET** palette to select a constructor of a .NET class from an assembly and create an instance of that class on execution. When you place this node on the block diagram, LabVIEW displays the **Select .NET Constructor** dialog box.
- Use the Property Node located on the **.NET** palette to get (read) and set (write) the properties associated with a .NET class.
- Use the Invoke Node located on the **.NET** palette to invoke methods associated with a .NET class.
- Use the Close Reference function located on the **.NET** palette to close all references to .NET objects when you no longer need the connection.
- Use the To More Generic Class function located on the **.NET** palette to upcast a .NET reference to its base class.
- Use the To More Specific Class function located on the **.NET** palette to downcast a .NET reference to its derived class.

LabVIEW as a .NET Client

When LabVIEW accesses the objects associated with .NET assemblies, it acts as a .NET client. Using LabVIEW as a .NET client involves the following three main steps.

1. Create a .NET object using a constructor and establish a reference to the object.
2. Wire the .NET object reference to a Property Node or Invoke Node and select a property or method.
3. Close the .NET object reference to close the connection to the object.

To access a .NET object, use the Constructor Node on the block diagram to create the required .NET object. Use the Constructor Node to select a class of object from an assembly. When you place a Constructor Node on the block diagram, LabVIEW displays the **Select .NET Constructor** dialog box, which lists all public assemblies in the GAC. If you want to select a private assembly, click the **Browse** button in the dialog box and navigate the file system for a private assembly. .NET assemblies are `.dll` and `.exe` file types. After you select a private assembly, the assembly appears in the **Assembly** pull-down menu in the **Select .NET Constructor** dialog box the next time you launch this dialog box.

When you select an assembly and class, the constructors for that class appear in the **Constructors** section of the **Select .NET Constructor** dialog box. Select the constructor and click the **OK** button to close this dialog box. LabVIEW displays the name of the class you selected on the Constructor Node.

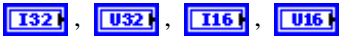
The Constructor Node is similar to the Automation Open function for ActiveX, except that a Constructor Node can specify initialization parameters for creating the object. You can wire the .NET server reference from the Constructor Node to a Property Node or Invoke Node and select a property or method from the shortcut menu. Use the Close Reference function to close the reference to the .NET object.

Refer to the *LabVIEW Help* for more information about creating .NET objects.

Data Type Mapping

LabVIEW converts the data types of the .NET property, method, and constructor parameters into LabVIEW data types so LabVIEW can read and interpret the data. LabVIEW displays data types it cannot convert as .NET refnums. The following table lists the .NET data types and the converted LabVIEW data type.

Table 19-1. .NET and LabVIEW Data Types

.NET Types	LabVIEW Types
System.Int32, System.UInt32, System.Int16, System.UInt16	
System.String	
System.Boolean	
System.Byte, System.Char, System.UByte	
System.Single, System.Double, System.Decimal	
System.Array	Displayed as an array of the corresponding type
Enumeration	 (sparse)
DateTime	
Any other .NET object	

Deploying .NET Applications

After you create a VI that includes .NET components, you can build that VI into an executable, LLB, or DLL.

Deploying an Executable

When you build applications that use .NET objects, you must copy the private assemblies the VIs use into the directory of the stand-alone application and make sure that the target computer has the .NET framework installed.

Deploying VIs

When you deploy the VIs, you must copy the private assemblies the VIs use to the directory of the top-level VI. Make sure all assemblies the VIs use are in the same directory structure of the top-level VI and make sure that the target computer has the .NET framework installed.

Deploying DLLs

When you deploy a LabVIEW-built DLL, you must copy the private assemblies the VIs use to the directory of the application that uses the LabVIEW-built DLL and make sure that the target computer has the .NET framework installed.

Configuring a .NET Client Application (Advanced)

.NET provides administrative capability to an application by using configuration files. A configuration file contains XML content and typically has a `.config` extension. To configure a LabVIEW .NET client application, you can provide a configuration file for a top-level VI or for a stand-alone application. Name the configuration file the name of the application with a `.config` extension, for example, `My App.vi.config` or `MyApp.exe.config`.

ActiveX Objects, Properties, Methods, and Events

ActiveX-enabled applications include objects that have exposed properties and methods that other applications can access. Objects can be visible to the users, such as buttons, windows, pictures, documents, and dialog boxes, or invisible to the user, such as application objects. You access an application by accessing an object associated with that application and setting a property or invoking a method of that object.

Refer to the *Manipulating Application and VI Settings* section of Chapter 17, *Programmatically Controlling VIs*, for more information about objects, properties, and methods in the LabVIEW environment.

Events are the actions taken on an object, such as clicking a mouse, pressing a key, or running out of memory. Whenever these actions occur to the object, the object sends an event to alert the ActiveX container along with the event-specific data.

Refer to the *ActiveX Events* section of this chapter for information about using the Register Event Callback node to handle ActiveX events.

ActiveX VIs, Functions, Controls, and Indicators

Use the following VIs, functions, controls, and indicators to access the objects, properties, methods, and events associated with other ActiveX-enabled applications:

- Use the automation refnum control to create a reference to an ActiveX object. Right-click this control on the front panel to select an object from the type library you want to access.
- Use the Automation Open function to open an ActiveX object.
- Use the ActiveX container to access and display an ActiveX object on the front panel. Right-click the container, select **Insert ActiveX Object** from the shortcut menu, and select the object you want to access.
- Use the Property Node to get (read) and set (write) the properties associated with an ActiveX object.
- Use the Invoke Node to invoke the methods associated with an ActiveX object.
- Use the Register Event Callback node to handle events that occur on an ActiveX object.
- Use the variant control and indicator to pass data to or from ActiveX controls. Refer to the *Handling Variant Data* section of Chapter 5, *Building the Block Diagram*, for more information about variant data.

LabVIEW as an ActiveX Client

When LabVIEW accesses the objects associated with another ActiveX-enabled application, it is acting as an ActiveX client. You can use LabVIEW as an ActiveX client in the following ways:

- Open an application, such as Microsoft Excel, create a document, and add data to that document.
- Embed a document, such as a Microsoft Word document or an Excel spreadsheet, on the front panel in a container.
- Place a button or other object from another application, such as a **Help** button that calls the other application help file, on the front panel.
- Link to an ActiveX control you built with another application.

LabVIEW accesses an ActiveX object with the automation refnum control or the ActiveX container, both of which are front panel objects. Use the automation refnum control to select an ActiveX object. Use the ActiveX container to select a displayable ActiveX object, such as a button or

document and place it on the front panel. Both objects appear as automation refnum controls on the block diagram.

Accessing an ActiveX-Enabled Application

To access an ActiveX-enabled application, use the automation refnum control on the block diagram to create a reference to an application. Wire the control to the Automation Open function, which opens the calling application. Use the Property Node to select and access the properties associated with the object. Use the Invoke Node to select and access the methods associated with the object. Close the reference to the object using the Close Reference function. Closing the reference removes the object from memory.

For example, you can build a VI that opens Microsoft Excel so it appears on the user's screen, creates a workbook, creates a spreadsheet, creates a table in LabVIEW, and writes that table to the Excel spreadsheet.

Refer to the Write Table To XL VI in the `examples\comm\ExcelExamples.llb` for an example of using LabVIEW as an Excel client.

Note Applications that include ActiveX custom interfaces appear with a icon. Click the icon to select an object for the custom interface. Refer to the *Support for Custom ActiveX Automation Interfaces* section of this chapter for more information about custom interfaces.

Inserting an ActiveX Object on the Front Panel

To insert an ActiveX object on the front panel, right-click the ActiveX container, select **Insert ActiveX Object** from the shortcut menu, and select the ActiveX control or document you want to insert. You can set the properties for an ActiveX object using the ActiveX Property Browser or property pages, or you can set the properties programmatically using the Property Node. Refer to the *Setting ActiveX Properties* section of this chapter for more information about setting ActiveX properties.

Use the Invoke Node to invoke the methods associated with the object.

For example, you can display a Web page on the front panel by using an ActiveX container to access the Microsoft Web Browser control, selecting the Navigate class of methods, selecting the URL method, and specifying the URL.

If you use the ActiveX container, you do not have to have to wire the automation refnum control on the block diagram to the Automation Open function or close the reference to the object using the Close Reference function. You can wire directly to the Invoke Node or Property Node because the ActiveX container embeds the calling application in LabVIEW. However, if the ActiveX container includes properties or methods that return other automation refnum references, you must close these additional references.

Design Mode for ActiveX Objects

Right-click an ActiveX container and select **Advanced»Design Mode** from the shortcut menu to display the container in design mode while you edit the VI. In design mode, events are not generated and event procedures do not run. The default mode is run mode, where you interact with the object as a user would.

Setting ActiveX Properties

After you open an ActiveX server or insert an ActiveX control or document, you can set the properties associated with that control or document using the ActiveX Property Browser, property pages, and the Property Node.

ActiveX Property Browser

Use the ActiveX Property Browser to view and set all the properties associated with an ActiveX control or document in an ActiveX container. To access the ActiveX Property Browser, right-click the control or document in the container on the front panel and select **Property Browser** from the shortcut menu. You also can select **Tools»Advanced»ActiveX Property Browser**. The ActiveX Property Browser is an easy way to set the properties of an ActiveX object interactively, but you cannot use the browser to set properties programmatically, and you can use the ActiveX Property Browser only with ActiveX objects in a container. The ActiveX Property Browser is not available in run mode or while a VI runs.

ActiveX Property Pages

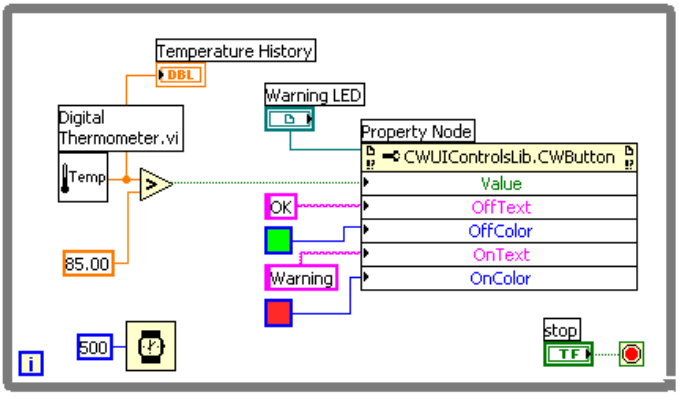
Many ActiveX objects include property pages, which organize the properties associated with the object on separate tabs. To access ActiveX property pages, right-click the object in the container on the front panel and select the name of the object from the shortcut menu.

Like the ActiveX Property Browser, ActiveX property pages are an easy way to set the properties of an ActiveX object interactively, but you cannot use them to set properties programmatically, and you can use property pages only with ActiveX objects in a container. Property pages are not available for all ActiveX objects. Property pages also are not available in Run mode or when the VI is running.

Property Nodes

Use the Property Node to set ActiveX properties programmatically. For example, if you use an ActiveX object to warn a user when a temperature exceeds a limit, use the Property Node to set the Value property of the object to specify the limit.

The following example changes the Value property of the CWButton ActiveX control, which is part of the National Instruments Measurement Studio User Interface ActiveX Library, when the temperature reaches 85 degrees Fahrenheit or higher.



In this case, the CWButton control acts as an LED, changes colors, and displays `Warning` when the temperature reaches the limit, which is the on state of the CWButton control.

Note In this example, you could use the ActiveX Property Browser or property pages to set the OffText, OffColor, OnText, and OnColor properties for the CWButton control because you do not need to set those properties programmatically. Refer to the *ActiveX Property Browser* and *ActiveX Property Pages* sections in this chapter for more information about the ActiveX Property Browser and property pages, respectively.

LabVIEW as an ActiveX Server

The LabVIEW application, VIs, and control properties and methods are available through ActiveX calls from other applications. Other ActiveX-enabled applications, such as Microsoft Excel, can request properties, methods, and individual VIs from LabVIEW, and LabVIEW acts as an ActiveX server.

For example, you can embed a VI graph in an Excel spreadsheet and, from the spreadsheet, enter data in the VI inputs and run the VI. When you run the VI, the data plot to the graph.

Refer to the `examples\comm\freqresp.xls` for an example of using LabVIEW properties and methods in an Excel spreadsheet.

Support for Custom ActiveX Automation Interfaces

If you are writing an ActiveX client that accesses properties and methods from an ActiveX server using LabVIEW, you can access custom interfaces exposed by the server. You do not need to use IDispatch to do so. However, the developer of the ActiveX server must make sure the parameters of the properties and methods in these custom interfaces have Automation (IDispatch) data types. The developer of the server must do so to expose multiple interfaces from one object, rather than through multiple objects. You still can use the interfaces in the LabVIEW environment. Refer to your server development programming environment documentation for more information about accessing custom interfaces.

Using Constants to Set Parameters in ActiveX VIs

Some parameters in ActiveX nodes take a discrete list of valid values. Select the descriptive name in the ring constant to set these parameter values. To access the ring constant when building an ActiveX VI, right-click the parameter of the node that accepts data values and select **Create Constant** from the shortcut menu. The selections available in the ring constant depend on the refnum passed to the node. Figures 19-1 and 19-2 show examples of using ring and numeric constants to set parameter values.

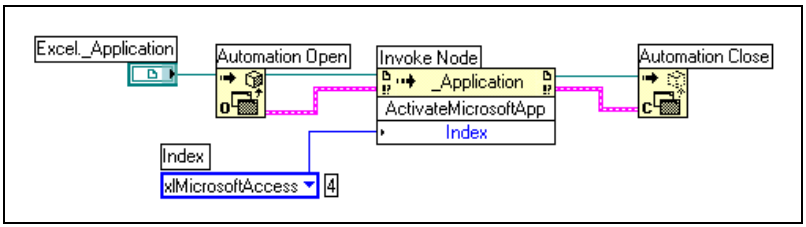


Figure 19-1. Setting a Data Value with a Ring Constant

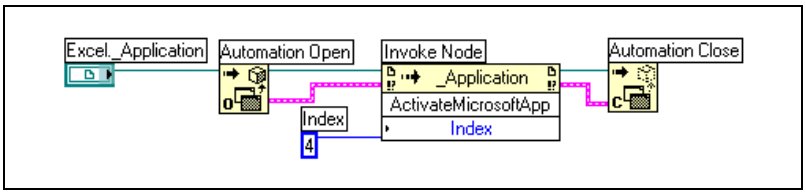


Figure 19-2. Setting a Data Value with a Numeric Constant

Parameters that accept data values have a small arrow to the left of the parameter name. To view the corresponding numeric data value, right-click the ring constant and select **Visible Items»Digital Display** from the shortcut menu.

The VIs in Figures 19-1 and 19-2 both access the Microsoft Excel application and execute a method. The **Index** parameter of the **ActivateMicrosoftApp** method has the following options: **MicrosoftWord**, **MicrosoftPowerPoint**, **MicrosoftMail**, **MicrosoftAccess**, **MicrosoftFoxPro**, **MicrosoftProject**, and **MicrosoftSchedulePlus**.

To identify the numeric value of the **Index** parameter that corresponds to the **MicrosoftAccess** option in Figure 19-1, select the **MicrosoftAccess** option from the pull-down menu in the ring constant. The numeric value of the currently selected option appears in a box next to the ring constant. Instead of using a ring constant, you can enter the numeric value of an option into a numeric constant, as shown in Figure 19-2.

ActiveX Events

To use ActiveX events in an application, you must register for the event and handle the event when it occurs. ActiveX event registration is similar to dynamic event registration as explained in the *Dynamic Event Registration* section of Chapter 9, *Event-Driven Programming*. However, the architecture of an ActiveX event VI is different than the architecture of an

event-handling VI as described in Chapter 9, *Event-Driven Programming*. The following components make up a typical ActiveX event VI:

- ActiveX object for which you want to generate an event.
- Register Event Callback node to specify and register for the type of event you want to generate.
- Callback VI that contains the code you write to handle the event you specify.

You can generate and handle events on ActiveX objects in a container or on ActiveX objects you specify by using an Automation Refnum. For example, you can call a Windows tree control from an ActiveX container and specify that you want to generate a Double Click event for the items displayed in the tree control.

The Register Event Callback node is a growable node capable of handling multiple events, similar to the Register For Events node.

When you wire an ActiveX object reference to the Register Event Callback node and specify the event you want to generate for that object, you are registering the ActiveX object for that event. After you register for the event, you create a callback VI that contains the code you write to handle the event.

Handling ActiveX Events

You must create a callback VI to handle events from ActiveX controls when the controls generate the registered events. The callback VI runs when the event occurs. To create a callback VI, right-click the **VI Ref** input of the Register Event Callback node and select **Create Callback VI** from the shortcut menu. LabVIEW creates a reentrant VI that contains the following elements:

- **Event common data** contains the following elements:
  - **Source** is a numeric control that specifies the source of the event, such as LabVIEW or ActiveX. A value of `1` indicates an ActiveX event.
  - **Type** specifies which event occurred. This is an enumerated type for user interface events and a 32-bit unsigned integer type for ActiveX and other event sources. For ActiveX events, the event type represents the method code, or ID, for the event registered.
  - **Time** is the time stamp in milliseconds that specifies when the event was generated.

- **CtlRef** is a reference to the ActiveX or Automation Refnum on which the event occurred.
- **Event Data** is a cluster of the parameters specific to the event the callback VI handles. LabVIEW determines the appropriate **Event Data** when you select an event from the Register Event Callback node. Refer to the *Notify and Filter Events* section of Chapter 9, *Event-Driven Programming*, for more information about notify and filter events.
- **Event Data Out** is a cluster of the modifiable parameters specific to the event the callback VI handles. This element is available only for filter events.
- (Optional) **User Parameter** is data LabVIEW passes to the user through the callback VI when the ActiveX object generates the event.

Note You can use an existing VI as a callback VI as long as the connector pane of the VI you intend to use matches the connector pane of the event data. National Instruments recommends that the callback VI be reentrant because if it is not, LabVIEW cannot call it simultaneously if ActiveX events occur multiple times.

20

Calling Code from Text-Based Programming Languages

You can call most standard shared libraries in LabVIEW using the Call Library Function Node. You also can call C code in LabVIEW using the Code Interface Node (CIN).

Refer to the *Call Library Function Nodes and Code Interface Nodes* section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about platform-specific considerations when calling external code. Refer to the *Using External Code in LabVIEW* manual for more information about calling code from text-based programming languages.

For more information…

Refer to the *LabVIEW Help* for more information about calling code from text-based programming languages.

Call Library Function Node

Use the Call Library Function Node to call most standard shared libraries or DLLs. With this function, you can create an interface in LabVIEW to call existing libraries or new libraries specifically written for use with LabVIEW. National Instruments recommends using the Call Library Function Node to create an interface to external code.

Code Interface Node

Use the CIN as an alternative method for calling source code written in C. The Call Library Function Node generally is easier to use than the CIN.

21

Formulas and Equations

When you want to use a complicated equation in LabVIEW, you do not have to wire together various arithmetic functions on the block diagram. You can develop equations in a familiar, mathematical environment and then integrate the equations into an application.

Use the Formula Node and Expression Node to perform mathematical operations in the LabVIEW environment. For more additional functionality, you can link to the mathematics application MATLAB.

For more information…

Refer to the *LabVIEW Help* for more information about using equations and the syntax to use, available functions and operators, and descriptions of possible errors.

Methods for Using Equations in LabVIEW

You can use the Formula Node, Expression Node, and MATLAB script node to perform mathematical operations on the block diagram.

Note You must have MATLAB installed on your computer to use the script node because the script node invokes the MATLAB script server to execute MATLAB scripts. Because LabVIEW uses ActiveX technology to implement the script node, it is available only on Windows. The script node is similar to the Formula Node, but it allows you to import an existing MATLAB script in ASCII form and run the script in LabVIEW. As with a Formula Node, you can pass data to and from the node.

Formula Nodes

The Formula Node is a convenient text-based node you can use to perform mathematical operations on the block diagram. You do not have to access any external code or applications, and you do not have to wire low-level arithmetic functions to create equations. In addition to text-based equation expressions, the Formula Node can accept text-based versions of if statements, while loops, for loops, and do loops, which are familiar to

C programmers. These programming elements are similar to what you find in C programming but are not identical.

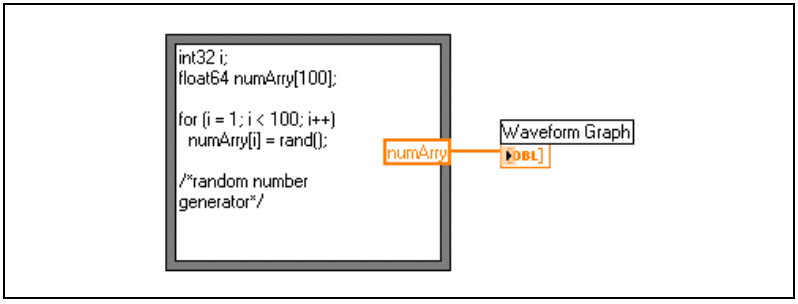
Formula Nodes are useful for equations that have many variables or are otherwise complicated and for using existing text-based code. You can copy and paste the existing text-based code into a Formula Node rather than recreating it graphically.

Formula Nodes use type checking to make sure that array indexes are numeric data and that operands to the bit operations are integer data. Formula Nodes also check to make sure array indexes are in range. For arrays, an out-of-range value defaults to zero, and an out-of-range assignment defaults to `nop` to indicate no operation occurs.

Formula Nodes also perform automatic type conversion.

Using the Formula Node

The Formula Node is a resizable box similar to the For Loop, While Loop, Case structure, Stacked Sequence structure, and Flat Sequence structure. However, instead of containing a subdiagram, the Formula Node contains one or more C-like statements delimited by semicolons, as in the following example. As with C, add comments by enclosing them inside a slash/asterisk pair (`/*comment*/`).



Refer to the Equations VI in the `examples\general\structs.llb` for an example of using a Formula Node.

Variables in the Formula Node

When you work with variables, remember the following points:

- There is no limit to the number of variables or equations in a Formula Node.
- No two inputs and no two outputs can have the same name, but an output can have the same name as an input.
- Declare an input variable by right-clicking the Formula Node border and selecting **Add Input** from the shortcut menu. You cannot declare input variables inside the Formula Node.
- Declare an output variable by right-clicking the Formula Node border and selecting **Add Output** from the shortcut menu. The output variable name must match either an input variable name or the name of a variable you declare inside the Formula Node.
- You can change whether a variable is an input or an output by right-clicking it and selecting **Change to Input** or **Change to Output** from the shortcut menu.
- You can declare and use a variable inside the Formula Node without relating it to an input or output wire.
- You must wire all input terminals.
- Variables can be floating-point numeric scalars, whose precision depends on the configuration of your computer. You also can use integers and arrays of numerics for variables.
- Variables cannot have units.

Expression Nodes

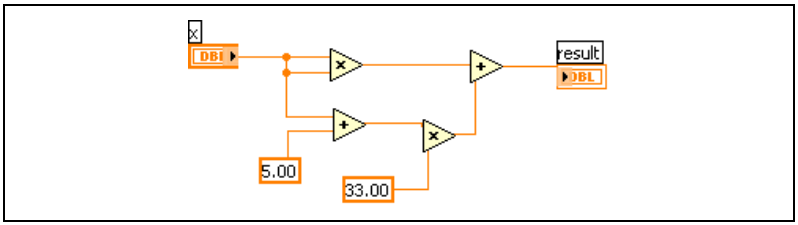
Use Expression Nodes to calculate expressions, or equations, that contain a single variable. Expression Nodes are useful when an equation has only one variable, but is otherwise complicated.

Expression Nodes use the value you pass to the input terminal as the value of the variable. The output terminal returns the value of the calculation.

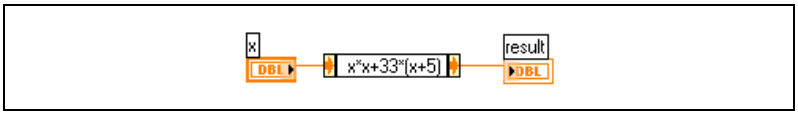
For example, consider this simple equation:

$$x \times x + 33 \times (x + 5)$$

The block diagram in the following figure uses Numeric functions to represent this equation.



Use an Expression Node, as shown in the following figure, to create a much simpler block diagram.



Polymorphism in Expression Nodes

The input terminal of an Expression Node is the same data type as the control or constant you wire to it. The output terminal is the same data type as the input terminal. The data type of the input can be any non-complex scalar number, array of non-complex scalar numbers, or cluster of non-complex scalar numbers. With arrays and clusters, the expression node applies the equation to each element of an input array or cluster.

MATLAB Script Nodes

Use the MATLAB script node to create, load, and edit MATLAB scripts on the block diagram.

You must have MATLAB installed to use the script node. If you already have a script written in MATLAB, you can import it into the script node.

You assign script node terminals as inputs or outputs for the variables in the script to pass values between MATLAB and LabVIEW. You can determine the function of a terminal by the way the equation is written. For example, if the script contains the assignment statement, $X = i + 3$, you can assign i as an input terminal to control how the script node calculates X, and you can assign X to an output terminal to retrieve the final result of the script calculation.

If you do not already have a script written, you can place a script node on the block diagram and create a script using MATLAB syntax. LabVIEW communicates with the script server engine, which is a program that runs the script. LabVIEW communicates and controls the script server engine using an industry-established protocol. A script server engine is installed with MATLAB.

Because of the nature of the MATLAB script language, the script node cannot determine the data type of terminals you created. You must assign a LabVIEW data type to each script node terminal.Table 21-1 shows LabVIEW data types and the corresponding data types in MATLAB.

Table 21-1. LabVIEW and MATLAB Data Types

LabVIEW Data Type	MATLAB Data Type
	Real
	String
	String
	Real Vector
	Real Matrix
	Complex
	Complex Vector
	Complex Matrix

Use the Conversion functions or the String/Array/Path Conversion functions to convert a LabVIEW data type to a data type that MATLAB supports.

Programming Suggestions for MATLAB Scripts

The following programming techniques make it easier to debug a script:

- Write the script and run it in MATLAB for testing and debugging purposes before you import it into LabVIEW.
- Verify the data types. When you create a new input or output, make sure the data type of the terminal is correct. You can use the Error In and Error Out functions to keep track of this.

- Create controls and indicators for the inputs and outputs to monitor the values the script node passes between LabVIEW and MATLAB. This allows you to pinpoint where a script node calculates a value incorrectly, if necessary.
- Take advantage of the error-checking parameters for debugging information. Create an indicator for the **error out** terminal on a script node so you can view error information at run time. Formula Nodes also show errors at compile time.

A

Organization of LabVIEW

This appendix describes the structure of the LabVIEW file system and the suggested locations for saving files.

Organization of the LabVIEW Directory Structure

This section describes the structure of the LabVIEW file system for Windows, Mac OS, and UNIX. LabVIEW installs driver software for GPIB, DAQ, VISA, IVI, Motion Control, and IMAQ hardware depending on platform availability. Refer to Chapter 3, *Configuring Measurement Hardware*, of the *LabVIEW Measurements Manual* for information about configuring your hardware.

The LabVIEW directory contains the following groupings after you complete the installation.

Libraries

- `user.lib`—Directory in which you can save controls and VIs you create. LabVIEW displays controls on **User Controls** palettes and VIs on **User Libraries** palettes. This directory does not change if you upgrade or uninstall LabVIEW. Refer to the *Adding VIs and Controls to the User and Instrument Drivers Subpalettes* section of Chapter 3, *LabVIEW Environment*, for more information about saving files in the `user.lib` directory.
- `vi.lib`—Contains libraries of built-in VIs, which LabVIEW displays in related groups on the **Functions** palette. Do not save files in the `vi.lib` directory because LabVIEW overwrites these files when you upgrade or reinstall.
- `instr.lib`—Contains instrument drivers used to control PXI, VXI, GPIB, serial, and computer-based instruments. When you install National Instruments instrument drivers and place them in this directory, LabVIEW adds them to the **Instrument Drivers** palette.

Structure and Support

- `menus`—Contains files LabVIEW uses to configure the structure of the **Controls** and **Functions** palettes.
- `resource`—Contains additional support files for the LabVIEW application. Do not save files into this directory because LabVIEW overwrites these files when you upgrade or reinstall.
- `project`—Contains files that become items on the LabVIEW **Tools** menu.
- `templates`—Contains templates for common VIs.
- `www`—Location of HTML files you can access through the Web Server.

Learning and Instruction

- `examples`—Contains example VIs. Select **Help»Find Examples** to browse or search the examples.

Documentation

- `manuals`—Contains documentation in PDF format. This folder does not contain the help files. Access the PDFs by selecting **Help»Search the LabVIEW Bookshelf**.
- `help`—Contains the help files. Access the *LabVIEW Help* by selecting **Help»VI, Function, & How-To Help**.

Mac OS

In addition to the above directories, Mac users have a shared libraries folder that contains support files for the LabVIEW application.

Suggested Location for Saving Files

LabVIEW installs the `vi.lib` and the `resource` directories for LabVIEW system purposes only. Do not save your files in these directories.

You can save your files in the following directories:

- `user.lib`—Any controls or VIs you want to display on the **User Controls** or **User Libraries** palettes. Refer to the *Adding VIs and Controls to the User and Instrument Drivers Subpalettes* section of Chapter 3, *LabVIEW Environment*, for more information about saving files in the `user.lib` directory.

Note Save subVIs in the `user.lib` directory only if they are portable, without modification, across projects. Paths to VIs in `user.lib` are relative to the `labview` directory. Paths to subVIs you save elsewhere are relative to the parent VI. Therefore, copying a VI from `user.lib` to modify it for a special case does not change the path to its subVIs located in `user.lib`.

- `instr.lib`—Any instrument driver VI you want to display on the **Instrument Drivers** palette.
- `project`—VIs you use to extend LabVIEW capabilities. VIs you store in this directory appear on the **Tools** menu.
- `www`—Location of HTML files you can access through the Web Server.
- `help`—Any VIs, PDFs, and `.hlp` files that you want to make available on the **Help** menu.
- `LabVIEW Data`— Any data files LabVIEW generates, such as `.lvm` or `.txt` files.

You also can create a directory anywhere on your hard drive to store LabVIEW files that you create. Refer to the *Saving VIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about creating directories.

B

Polymorphic Functions

Functions are polymorphic to varying degrees—none, some, or all of their inputs can be polymorphic. Some function inputs accept numerics or Boolean values. Some accept numerics or strings. Some accept not only scalar numerics but also arrays of numerics, clusters of numerics, arrays of clusters of numerics, 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 numerics. Refer to the *Polymorphic Units in LabVIEW* Application Note for more information about creating and using polymorphic units.

For more information…

Refer to the *LabVIEW Help* for more information about polymorphic functions.

Numeric Conversion

You can convert any numeric representation to any other numeric representation. When you wire two or more numeric inputs of different representations to a function, the function usually returns output in the larger or wider format. The functions coerce the smaller representations to the widest representation before execution and LabVIEW places a coercion dot on the terminal where the conversion takes place.

Some functions, such as Divide, Sine, and Cosine, always produce floating-point output. If you wire integers to their inputs, these functions convert the integers to double-precision, floating-point numbers before performing the calculation.

For floating-point, scalar quantities, it is usually best to use double-precision, floating-point numbers. Single-precision, floating-point numbers save little or no run time and overflow much more easily. The analysis libraries, for example, use double-precision, floating-point numbers. You should only use extended-precision, floating-point numbers when necessary. The performance and precision of extended-precision arithmetic varies among the platforms. Refer to the *Undefined or*

Unexpected Data section of Chapter 6, *Running and Debugging VIs*, for more information about floating-point overflow.

For integers, it is usually best to use a 32-bit signed integer.

If you wire an output to a destination that has a different numeric representation, LabVIEW converts the data according to the following rules:

- **Signed or unsigned integer to floating-point number**—Conversion is exact, except for 32-bit integers to single-precision, floating-point numbers. In this case, LabVIEW reduces the precision from 32 bits to 24 bits.
- **Floating-point number to signed or unsigned integer**—LabVIEW moves out-of-range values to the integer's minimum or maximum value. Most integer objects, such as the iteration terminal of a For Loop, round floating-point numbers. LabVIEW rounds a fractional part of 0.5 to the nearest even integer. For example, LabVIEW rounds 6.5 to 6 rather than 7.
- **Integer to integer**—LabVIEW does not move out-of-range values to the integer's minimum or maximum value. If the source is smaller than the destination, LabVIEW extends the sign of a signed source and places zeros in the extra bits of an unsigned source. If the source is larger than the destination, LabVIEW copies only the least significant bits of the value.

Polymorphism for Numeric Functions

The arithmetic functions take numeric input data. With some exceptions noted in the function descriptions, the output has the same numeric representation as the input or, if the inputs have different representations, the output is the wider of the inputs.

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.

You cannot use the Multiply function to do matrix multiplication. If you use the Multiply function with two matrices, LabVIEW takes the first number in the first row of the first matrix, multiplies it by the first number in the first row of the second matrix, and so on.

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 B-1 shows the possible polymorphic combinations of the Add function.

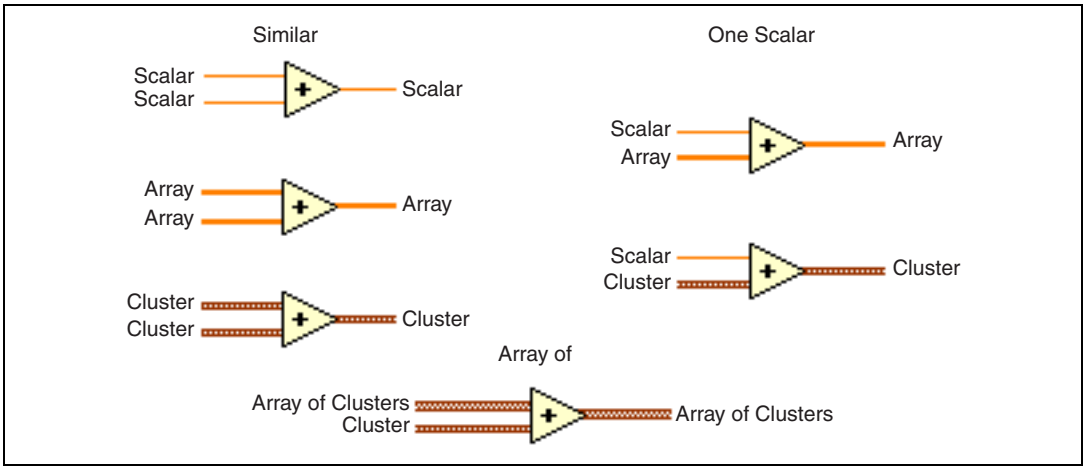


Figure B-1. Polymorphic Combinations of the Add Function

Polymorphism for Boolean Functions

The logical functions take either Boolean or numeric input data. If the input is numeric, LabVIEW performs a bit-wise 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 long integer, and the output is a long integer.

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:

> *Logical type* = Boolean scalar OR numeric scalar OR array [*logical type*] OR cluster [*logical types*]

except that complex numbers and arrays of arrays are not allowed.

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 B-2 shows some combinations of Boolean values for the AND function.

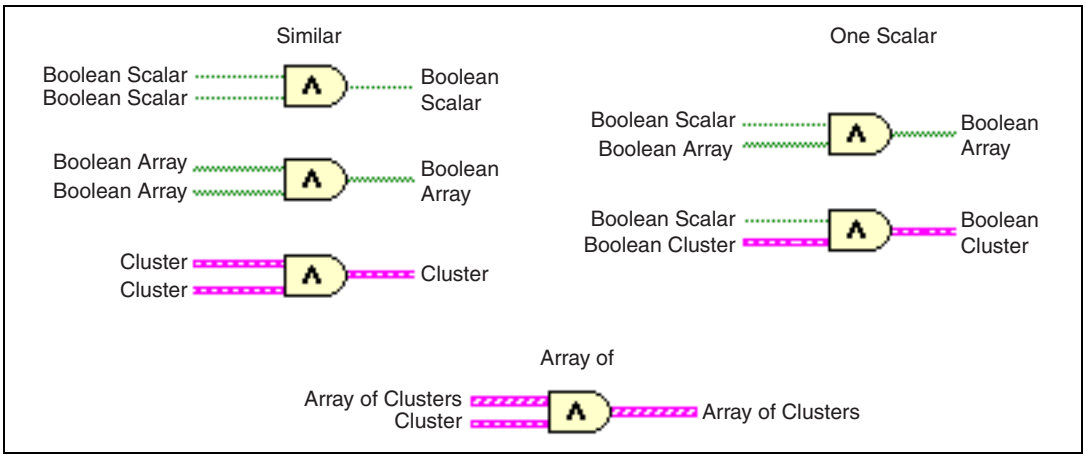


Figure B-2. Boolean Combinations for the AND Function

Polymorphism for Array Functions

Most of the array functions accept *n*-dimensional arrays of any type. However, the wiring diagrams in the function descriptions show numeric arrays as the default data type.

Polymorphism for String Functions

String Length, To Upper Case, To Lower Case, Reverse String, and Rotate String accept strings, clusters and arrays of strings, and arrays of clusters. To Upper Case and To Lower Case also accept numbers, clusters of numbers, and arrays of numbers, interpreting them as ASCII codes for characters. Width and precision inputs must be scalar.

Polymorphism for String Conversion Functions

The Path To String and String To Path functions are polymorphic. That is, they work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Polymorphism for Additional String to Number Functions

To Decimal, To Hex, To Octal, To Engineering, To Fractional, and To Exponential accept clusters and arrays of numbers and produce clusters and arrays of strings. From Decimal, From Hex, From Octal, and From

Exponential/Fract/Sci accept clusters and arrays of strings and produce clusters and arrays of numbers. Width and precision inputs must be scalar.

Polymorphism for Cluster Functions

The Bundle and Unbundle functions do not show the data type for their individual input or output terminals until you wire objects to these terminals. When you wire them, these terminals look similar to the data types of the corresponding front panel control or indicator terminals.

Polymorphism for Comparison Functions

The functions Equal?, Not Equal?, and Select take inputs of any type, as long as the inputs are the same type.

The functions Greater or Equal?, Less or Equal?, Less?, Greater?, Max & Min, and In Range? take inputs of any type except complex, path, or refnum, as long as the inputs are the same type. You can compare numbers, strings, Booleans, arrays of strings, clusters of numbers, clusters of strings, and so on. However, you cannot compare a number to a string or a string to a Boolean, and so on.

The functions that compare values to zero accept numeric scalars, clusters, and arrays of numbers. These functions output Boolean values in the same data structure as the input.

The Not A Number/Path/Refnum function accepts the same input types as functions that compare values to zero. This function also accepts paths and refnums. Not A Number/Path/Refnum outputs Boolean values in the same data structure as the input.

The functions Decimal Digit?, Hex Digit?, Octal Digit?, Printable?, and White Space? accept a scalar string or number input, clusters of strings or non-complex numbers, arrays of strings or non-complex numbers, and so on. The output consists of Boolean values in the same data structure as the input.

The function Empty String/Path? accepts a path, a scalar string, clusters of strings, arrays of strings, and so on. The output consists of Boolean values in the same data structure as the input.

You can use the Equal?, Not Equal?, Not A Number/Path/Refnum?, Empty String/Path?, and Select functions with paths and refnums, but no other Comparison functions accept paths or refnums as inputs.

Comparison functions that accept arrays and clusters normally return Boolean arrays and clusters of the same structure. If you want the function to return a single Boolean value, right-click the function and select **Comparison Mode»Compare Aggregates** from the shortcut menu. Refer to the *Comparing Arrays and Clusters* section of Appendix C, *Comparison Functions*, for more information about how the function compares aggregates.

Polymorphism for Log Functions

The Logarithmic functions take numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These 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*]

except that arrays of arrays are not allowed.

Arrays can be any size and can have any number of dimensions. Clusters can have any number of elements. The output type is of the same numeric representation as the input, and the functions operate on each element of the cluster or array. Refer to the *Polymorphism for Numeric Functions* section of this appendix for more information about two-input polymorphic functions. Allowable input type combinations for the two-input Logarithmic functions include the following:

- **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 a numeric array or cluster, and the output is an array or cluster.

C

Comparison Functions

Use the Comparison functions to compare Boolean values, strings, numerics, arrays, and clusters. Most Comparison functions test one input or compare two inputs and return a Boolean value.

For more information…

Refer to the *LabVIEW Help* for more information about Comparison functions.

Comparing Boolean Values

The Comparison functions treat the Boolean value TRUE as greater than the Boolean value FALSE.

Comparing Strings

LabVIEW compares strings based on the numerical equivalent of the ASCII characters. For example, `a` (with a decimal value of 97) is greater than `A` (65), which is greater than the numeral `0` (48), which is greater than the space character (32). LabVIEW compares characters one by one from the beginning of the string until an inequality occurs, at which time the comparison ends. For example, LabVIEW evaluates the strings `abcd` and `abef` until it finds `c`, which is less than the value of `e`. The presence of a character is greater than the absence of one. Thus, the string `abcd` is greater than `abc` because the first string is longer.

The functions that test the category of a string character, such as the Decimal Digit? and Printable? functions, evaluate only the first character of the string.

Comparing Numerics

The Comparison functions convert numeric values to the same representation before comparing them. Comparisons with one or two inputs having the value Not a Number (NaN) return a value that indicates inequality. Refer to the *Undefined or Unexpected Data* section of Chapter 6, *Running and Debugging VIs*, for more information about the NaN value.

Comparing Arrays and Clusters

Some Comparison functions have two modes for comparing arrays or clusters of data. In Compare Aggregates mode, if you compare two arrays or clusters, the function returns a single Boolean value. In Compare Elements mode, the function compares the elements individually and returns an array or cluster of Boolean values.

In Compare Aggregates mode, the string comparison and array comparison operations follow exactly the same process—the Comparison function treats the string as an array of ASCII characters.

Right-click a Comparison function and select **Comparison Mode» Compare Elements** or **Comparison Mode»Compare Aggregates** from the shortcut menu to change the mode of the function. Some Comparison functions operate only in Compare Aggregates mode, so the shortcut menu items do not appear.

Arrays

When comparing multidimensional arrays, each array wired to the function must have the same number of dimensions. The Comparison functions that operate only in Compare Aggregates mode compare arrays the same way they compare strings—one element at a time starting with the first element until they encounter an inequality.

Compare Elements Mode

In Compare Elements mode, Comparison functions return an array of Boolean values of the same dimensions as the input arrays. Each dimension of the output array is the size of the smaller of the two input arrays in that dimension. Along each dimension such as a row, column, or page, the functions compare corresponding element values in each input array to produce the corresponding Boolean value in the output array.

Compare Aggregates Mode

In Compare Aggregates mode, comparison functions return a single Boolean result after comparing the elements in an array. Comparison functions consider elements later in the array secondary to elements earlier in the array. Therefore, the function performs the following steps to determine the result of the comparison:

- The function compares corresponding elements in each input array, starting at the beginning of the array.
- If the corresponding elements are *not* equal, the function stops—it returns the result of this comparison.
- If the corresponding elements are equal, the Comparison function processes the next pair of values, until it finds an inequality or reaches the end of one of the input arrays.
- If all values in the input arrays are equal but one array has extra elements at the end, the longer array is considered greater than the shorter one. For example, a Comparison function considers the array `[1,2,3,2]` to be greater than the array `[1,2,3]`.

Clusters

Clusters you compare must include the same number of elements, each element in the clusters must be of compatible types, and the elements must be in the same cluster order. For example, you can compare a cluster of a DBL and a string to a cluster of an I32 and a string.

Compare Elements Mode

In Compare Elements mode, Comparison functions return a cluster of Boolean elements, one for each corresponding element in the input clusters.

Compare Aggregates Mode

In Compare Aggregates mode, Comparison functions return a single Boolean value. The function compares corresponding elements until it finds an inequality, which determines the result. The function considers the two clusters equal only if all elements are equal.

Use Compare Aggregates mode on clusters if you are comparing two records containing sorted data, where elements later in the cluster are considered secondary to elements earlier in the cluster. For example, if you compare a cluster containing two strings, last name followed by first name, the function would compare the first name fields only if the last name fields matched.

D

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/custed for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. 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.

Glossary

Symbol	Prefix	Value
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}

Numbers/Symbols

Δ	Delta; difference. Δ*x* denotes the value by which *x* changes from one index to the next.
π	Pi.
∞	Infinity.
1D	One-dimensional.
2D	Two-dimensional.
3D	Three-dimensional.

A

A	Amperes.
absolute coordinates	Picture coordinates relative to the origin (0, 0) of the picture indicator.
absolute path	File or directory path that describes the location relative to the top level of the file system.
AC	Alternating current.
active window	Window that is currently set to accept user input, usually the frontmost window. The title bar of an active window is highlighted. Make a window active by clicking it or by selecting it from the **Windows** menu.

application software	Application created using the LabVIEW Development System and executed in the LabVIEW Run-Time System environment.
array	Ordered, indexed list of data elements of the same type.
artificial data dependency	Condition in a dataflow programming language in which the arrival of data, rather than its value, triggers execution of a node.
ASCII	American Standard Code for Information Interchange.
auto-indexing	Capability of loop structures to disassemble and assemble arrays at their borders. As an array enters a loop with auto-indexing enabled, the loop automatically disassembles it extracting scalars from ID arrays, ID arrays extracted from 2D arrays, and so on. Loops assemble data into arrays as data exit the loop in the reverse order.
autoscaling	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.
breakpoint	Pause in execution used for debugging.
Breakpoint tool	Tool to set a breakpoint on a VI, node, or wire.
broken **Run** button	Button that replaces the **Run** button when a VI cannot run because of errors.
broken VI	VI that cannot or run because of errors; signified by a broken arrow in the broken **Run** button.
buffer	Temporary storage for acquired or generated data.
byte stream file	File that stores data as a sequence of ASCII characters or bytes.

C

case	One subdiagram of a Case structure.
Case structure	Conditional branching control structure, that executes one of its subdiagrams based on the input to the Case structure. It is the combination of the IF, THEN, ELSE, and CASE statements in control flow languages.
channel	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. 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 and earlier versions, but is integral to every measurement you take in NI-DAQmx. In Traditional NI-DAQ, 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. 3. Switch—a switch channel represents any connection point on a switch. It may 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.
channel name	Unique name given to a channel configuration in the DAQ Channel Wizard.
chart	2D display of one or more plots in which the display retains a history of previous data, up to a maximum that you define. The chart receives the data and updates the display point by point or array by array, retaining a certain number of past points in a buffer for display purposes. *See also* scope chart, strip chart, and sweep chart.
checkbox	Small square box in a dialog box you can select or clear. Checkboxes generally are associated with multiple options that you can set. You can select more than one checkbox.
CIN	*See* Code Interface Node (CIN).

class	A category containing properties, methods, and events. Classes are arranged in a hierarchy with each class inheriting the properties and methods associated with the class in the preceding level.
cloning	To make a copy of a control or another object by clicking it while pressing the <Ctrl> key and dragging the copy to its new location. **(Mac OS)** Press the <Option> key. **(Sun)** Press the <Meta> key. **(Linux)** Press the <Alt> key. **(UNIX)** You also can clone an object by clicking the object with the middle mouse button and then dragging the copy to its new location.
cluster	A set of ordered, unindexed data elements of any data type, including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
Code Interface Node (CIN)	CIN. Special block diagram node through which you can link text-based code to a VI.
coercion	Automatic conversion LabVIEW performs to change the numeric representation of a data element.
coercion dot	Appears on a block diagram node to alert you that you have wired data of two different numeric data types together. Also appears when you wire any data type to a variant data type.
Color Copying tool	Copies colors for pasting with the Coloring tool.
Coloring tool	Tool to set foreground and background colors.
compile	Process that converts high-level code to machine-executable code. LabVIEW compiles VIs automatically before they run for the first time after you create or edit alteration.
conditional terminal	Terminal of a While Loop that contains a Boolean value that determines if the VI performs another iteration.
configuration utility	Refers to Measurement & Automation Explorer on Windows and to the NI-DAQ Configuration Utility on Mac OS.
connector	Part of the VI or function node that contains input and output terminals. Data pass to and from the node through a connector.

connector pane	Region in the upper right corner of a front panel or block diagram window that displays the VI terminal pattern. It defines the inputs and outputs you can wire to a VI.
constant	*See* universal constant *and* user-defined constant.
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.
control flow	Programming system in which the sequential order of instructions determines execution order. Most text-based programming languages are control flow languages.
Controls palette	Palette that contains front panel controls, indicators, and decorative objects.
conversion	Changing the type of a data element.
count terminal	Terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.
current VI	VI whose front panel, block diagram, or Icon Editor is the active window.
curve in 3D	Special parametric plot ($x(t)$, $y(t)$, $z(t)$), where the parameter t runs over a given interval.

D

D	Delta; Difference. Δx denotes the value by which x changes from one index to the next.
DAQ	*See* data acquisition (DAQ).
DAQ Channel Wizard	Utility that guides you through naming and configuring DAQ analog and digital channels. Available in the **(Windows)** Data Neighborhood of Measurement & Automation Explorer or **(Mac OS)** DAQ Channel Wizard.

data acquisition (DAQ)	1. Acquiring and measuring analog or digital electrical signals from sensors, transducers, and test probes or fixtures. 2. Generating analog or digital electrical signals.
data dependency	Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *See also* artificial data dependency.
data flow	Programming system that consists of executable nodes that execute only when they receive all required input data and produce output automatically when they execute. LabVIEW is a dataflow system.
datalog file	File that stores data as a sequence of records of a single, arbitrary data type that you specify when you create the file. Although all the records in a datalog file must be a single type, that type can be complex. For example, you can specify that each record is a cluster that contains a string, a number, and an array.
datalogging	Generally, to acquire data and simultaneously store it in a disk file. LabVIEW File I/O VIs and functions can log data.
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.
directory	Structure for organizing files into convenient groups. A directory is like an address that shows the location of files. A directory can contain files or subdirectories of files.
discrete	Having discontinuous values of the independent variable, usually time.
dithering	Addition of Gaussian noise to an analog input signal. By applying dithering and then averaging the input data, you can effectively increase the resolution by another one-half bit.
DLL	Dynamic Link Library.

drag — To use the cursor on the screen to select, move, copy, or delete objects.

drive — Letter in the range `a–z` followed by a colon (`:`), to indicate a logical disk drive.

driver — Software unique to the device or type of device, and includes the set of commands the device accepts.

E

empty array — Array that has zero elements but has a defined data type. For example, an array that has a numeric control in its data display window but has no defined values for any element is an empty numeric array.

error in — Error structure that enters a VI.

error message — Indication of a software or hardware malfunction or of an unacceptable data entry attempt.

error out — The error structure that leaves a VI.

error structure — Consists of a Boolean status indicator, a numeric code indicator, and a string source indicator.

event — Condition or state of an analog or digital signal.

Event Data Node — Node attached to the left and right sides of an Event structure indicating the available data for the event you configured that case to handle. If you configure a single case to handle multiple events, only the data that is common to all handled event types is available.

execution highlighting — Debugging technique that animates VI execution to illustrate the data flow in the VI.

external trigger — Voltage pulse from an external source that triggers an event, such as A/D conversion.

F

FIFO — First-in-first-out memory buffer. The first data stored is the first data sent to the acceptor.

file refnum — *See* refnum.

filter events	Allow control over how the user interface behaves.
filtering	Type of signal conditioning that allows you to filter unwanted signals from the signal you are trying to measure.
Flat Sequence structure	Program control structure that executes its subdiagrams in numeric order. Use this structure to force nodes that are not data dependent to execute in the order you want. The Flat Sequence structure displays all the frames at once and executes the frames from left to right until the last frame executes.
flattened data	Data of any type that has been converted to a string, usually for writing the data to a file.
For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to text-based code: `For i = 0 to n – 1, do...`
Formula Node	Node that executes equations you enter as text. Especially useful for lengthy equations too cumbersome to build in block diagram form.
frame	Subdiagram of a Flat or Stacked Sequence structure.
free label	Label on the front panel or block diagram that does not belong to any other object.
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.
global variable	Accesses and passes data among several VIs on a block diagram.
glyph	Small picture or icon.

GPIB — *See* General Purpose Interface Bus.

graph — 2D display of one or more plots. A graph receives and plots data as a block.

graph control — Front panel object that displays data in a Cartesian plane.

group — Collection of input or output channels or ports that you define. Groups can contain analog input, analog output, digital input, digital output, or counter/timer channels. A group can contain only one type of channel. Use a task ID number to refer to a group after you create it. You can define up to 16 groups at one time.

To erase a group, pass an empty channel array and the group number to the group configuration VI. You do not need to erase a group to change its membership. If you reconfigure a group whose task is active, LabVIEW clears the task and returns a warning. LabVIEW does not restart the task after you reconfigure the group.

H

handle — Pointer to a pointer to a block of memory that manages reference arrays and strings. An array of strings is a handle to a block of memory that contains handles to strings.

hex — Hexadecimal. Base-16 number system.

Hierarchy window — Window that graphically displays the hierarchy of VIs and subVIs.

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.

IEEE — Institute for Electrical and Electronic Engineers.

indicator — Front panel object that displays output, such as a graph or LED.

`Inf` — Digital display value for a floating-point representation of infinity.

inplace	The condition in which two or more terminals, such as error I/O terminals or shift registers, use the same memory space.
instrument driver	VI that controls a programmable instrument.
integer	Any of the natural numbers, their negatives, or zero.
intensity map/plot	Method of displaying three dimensions of data on a 2D plot with the use of color.
IP	Internet protocol.
iteration terminal	Terminal of a For Loop or While Loop that contains the current number of completed iterations.

L

label	Text object used to name or describe objects or regions on the front panel or block diagram.
Labeling tool	Tool to create labels and enter text into text windows.
LabVIEW	Laboratory Virtual Instrument Engineering Workbench. LabVIEW is a graphical programming language that uses icons instead of lines of text to create programs.
LabVIEW system time	The date and time that LabVIEW uses as a reference for absolute time. LabVIEW system time is defined as 12:00 a.m., January 1, 1904, Universal time.
LED	Light-emitting diode.
legend	Object a graph or chart owns to displays the names and plot styles of plots on that graph or chart.
library	*See* VI library.
listbox	Box within a dialog box that lists all available choices for a command. For example, a list of filenames on a disk.
LLB	VI Library.
local variable	Variable that enables you to read or write to one of the controls or indicators on the front panel of a VI.

M

matrix	A rectangular array of numbers or mathematical elements that represent the coefficients in a system of linear equations.
Measurement & Automation Explorer	The standard National Instruments hardware configuration and diagnostic environment for Windows.
memory buffer	*See* buffer.
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.
method	A procedure that is executed when an object receives a message. A method is always associated with a class.
multithreaded application	Application that runs several different threads of execution independently. On a multiple processor computer, the different threads might be running on different processors simultaneously.

N

`NaN`	Digital display value for a floating-point representation of *not a number*. Typically the result of an undefined operation, such as log(–1).
NI-DAQ	Driver software included with all NI measurement devices. NI-DAQ is an extensive library of VIs and functions you can call from an application development environment (ADE), such as LabVIEW, to program all the features of an NI measurement device, such as configuring, acquiring, and generating data from, and sending data to the device.
NI-DAQmx	The new 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 measurement tasks, channels, and scales; increased performance; expanded functionality; and a simpler application programming interface (API).
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.

notify events	Tell LabVIEW that a user action has already occurred, such as when a user changes the value of a control.
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, nodes, wires, and imported pictures.
Object Shortcut Menu tool	Tool to access a shortcut menu for an object.
OLE	Object Linking and Embedding.
one-dimensional	Having one dimension, as in the case of an array that has only one row of elements.
Operating tool	Tool to enter data into controls operate them.

P

palette	Display of icons that represent possible options.
panel window	VI window that contains the front panel, the toolbar, and the icon and connector panes.
picture	Series of graphics instructions that a picture indicator uses to create a picture.
picture indicator	General-purpose indicator for displaying pictures that can contain lines, circles, text, and other types of graphic shapes.
pixel	Smallest unit of a digitized picture.
pixmap	Standard format for storing pictures in which a color value represents each pixel. A bitmap is a black and white version of a pixmap.
plot	Graphical representation of an array of data shown either on a graph or a chart.

point	Cluster that contains two 16-bit integers that represent horizontal and vertical coordinates.
polymorphism	Ability of a node to automatically adjust to data of different representation, type, or structure.
Positioning tool	Tool to move and resize objects.
PPC	Program-to-program communication.
probe	Debugging feature for checking intermediate values in a VI.
Probe tool	Tool to create probes on wires.
Property Node	Sets or finds the properties of a VI or application.
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.

R

race condition	Occurs when two or more pieces of code that execute in parallel change the value of the same shared resource, typically a global or local variable.
range	Region between the limits within which a quantity is measured, received, or transmitted. Expressed by stating the lower and upper range values.
rectangle	Cluster that contains four 16-bit integers. The first two values describe the vertical and horizontal coordinates of the top left corner. The last two values describe the vertical and horizontal coordinates of the bottom right corner.
refnum	Reference number. An identifier that LabVIEW associates with a file you open. Use the refnum to indicate that you want a function or VI to perform an operation on the open file.
relative coordinates	Picture coordinates relative to the current location of the pen.
representation	Subtype of the numeric data type, of which there are 8-bit, 16-bit, and 32-bit signed and unsigned integers, as well as single-, double-, and extended-precision floating-point numbers.

resizing circles or handles	Circles or handles that appear on the borders of an object to indicate the points where you can resize the object.
ring control	Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

S

sample	Single analog or digital input or output data point.
scalar	Number that a point on a scale can represent. A single value as opposed to an array. Scalar Booleans and clusters are explicitly singular instances of their respective data types.
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.
scope chart	Numeric indicator modeled on the operation of an oscilloscope.
Scrolling tool	Tool to move through windows.
SCXI	Signal Conditioning eXtensions for Instrumentation. The National Instruments product line for conditional low-level signals within an external chassis near sensors, so only high-level signals in a noisy environment are sent to DAQ devices.
sequence local	Terminal to pass data between the frames of a Stacked Sequence structure.
sequence structure	See Flat Sequence structure or Stacked Sequence structure.
shared library	A file containing executable program modules that any number of different programs can use to perform some function. Shared libraries are useful when you want to share the functionality of the VIs you build with other developers.
shift register	Optional mechanism in loop structures to pass the value of a variable from one iteration of a loop to a subsequent iteration. Shift registers are similar to static variables in text-based programming languages.
shortcut menu	Menu accessed by right-clicking an object. Menu items pertain to that object specifically.
slider	Moveable part of slide controls and indicators.

Stacked Sequence structure	Program control structure that executes its subdiagrams in numeric order. Use this structure to force nodes that are not data dependent to execute in the order you want. The Stacked Sequence structure displays each frame so you see only one frame at a time and executes the frames in order until the last frame executes.
string	Representation of a value as text.
string controls and indicators	Front panel objects to manipulate and display text.
strip chart	Numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.
structure	Program control element, such as a Flat Sequence structure, Stacked Sequence structure, Case structure, For Loop, or While Loop.
subdiagram	Block diagram within the border of a structure.
subVI	VI used on the block diagram of another VI. Comparable to a subroutine.
sweep chart	Numeric indicator modeled on the operation of an oscilloscope. It is similar to a scope chart, except that a line sweeps across the display to separate old data from new data.
syntax	Set of rules to which statements must conform in a particular programming language.

T

TCP/IP	Transmission Control Protocol/Internet Protocol. A standard format for transmitting data in packets from one computer to another. The two parts of TCP/IP are TCP, which deals with the construction of data packets, and IP, which routes them from computer to computer.
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.

Tools palette	Palette that contains tools you can use to edit and debug front panel and block diagram objects.
top-level VI	VI at the top of the VI hierarchy. This term distinguishes the VI from its subVIs.
tunnel	Data entry or exit terminal on a structure.
two-dimensional	Having two dimensions, as in the case of an array that has several rows and columns.
type definition	Master copy of a custom object that several VIs can use.

U

UDP	User Datagram Protocol.
universal constant	Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, for example, π.
URL	Uniform resource locator. A logical address that identifies a resource on a server, usually on the Web. For example, `http://www.ni.com/` is the URL for the National Instruments Web site.
user-defined constant	Block diagram object that emits a value you set.

V

vector	1D array.
VI	*See* virtual instrument (VI).
VI class	A reference to a virtual instrument that allows access to VI properties and methods.
VI library	Special file that contains a collection of related VIs for a specific use.
VI Server	Mechanism for controlling VIs and LabVIEW applications programmatically, locally and remotely.
virtual instrument (VI)	Program in LabVIEW that models the appearance and function of a physical instrument.

Virtual Instrument Software Architecture	VISA. Single interface library for controlling GPIB, VXI, RS-232, and other types of instruments.
VISA	*See* Virtual Instrument Software Architecture.
VXI	VME eXtensions for Instrumentation (bus).

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 is met.
wire	Data path between nodes.
wire bend	Point where two wire segments join.
wire branch	Section of wire that contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions between.
wire junction	Point where three or more wire segments join.
wire segment	Single horizontal or vertical piece of wire.
wire stubs	Truncated wires that appear next to unwired terminals when you move the Wiring tool over a VI or function node.
Wiring tool	Tool to define data paths between terminals.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z