# LabVIEW™

## Getting Started with LabVIEW for the *FIRST* Robotics Competition

October 2009
372600B-01

NATIONAL INSTRUMENTS™

**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 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

© 2008–2009 National Instruments Corporation. All rights reserved.

# 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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

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

## Trademarks

National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on `ni.com/legal` for more information about National Instruments trademarks.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

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

## Patents

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

## 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

## About This Manual

Conventions ........................................................ xi

## Chapter 1 Introduction to LabVIEW

LabVIEW VI Templates and Example VIs ........................................................ 1-1
- LabVIEW VI Templates ........................................................ 1-1
- LabVIEW Example VIs ........................................................ 1-2

Related Documentation ........................................................ 1-2
- LabVIEW Help ........................................................ 1-2
- LabVIEW Manuals ........................................................ 1-3
- FRC-Specific Resources ........................................................ 1-3

## Chapter 2 Introduction to Virtual Instruments

Front Panel ........................................................ 2-2
Block Diagram ........................................................ 2-2
- Terminals ........................................................ 2-3
- Nodes ........................................................ 2-4
- Wires ........................................................ 2-4
- Structures ........................................................ 2-4

Icon and Connector Pane ........................................................ 2-5
Using and Customizing VIs and SubVIs ........................................................ 2-5

## Chapter 3 LabVIEW Environment

Getting Started Window ........................................................ 3-1
Context Help Window ........................................................ 3-2
Project Explorer Window ........................................................ 3-2
Navigation Window ........................................................ 3-3
Controls Palette ........................................................ 3-3
Functions Palette ........................................................ 3-3
Navigating the Controls and Functions Palettes ........................................................ 3-4
Tools Palette ........................................................ 3-4

Menus and Toolbars ........ 3-4
Shortcut Menus ........ 3-5
Shortcut Menus in Run Mode ........ 3-5
VI Toolbar ........ 3-5
Project Explorer Window Toolbars ........ 3-5
Customizing Your Work Environment ........ 3-6

## Chapter 4
## Building the Front Panel

Front Panel Controls and Indicators ........ 4-1
Numeric Controls and Indicators ........ 4-1
Boolean Controls and Indicators ........ 4-2
String Controls and Indicators ........ 4-2
Configuring Front Panel Objects ........ 4-3
Changing Controls to Indicators and Indicators to Controls ........ 4-3
Replacing Front Panel Objects ........ 4-3
Configuring the Front Panel ........ 4-4
Coloring Objects ........ 4-4
Aligning and Distributing Objects ........ 4-5
Grouping and Locking Objects ........ 4-5
Resizing Objects ........ 4-5
Adding Space to the Front Panel without Resizing the Window ........ 4-6
Labeling ........ 4-6
Designing User Interfaces ........ 4-7

## Chapter 5
## Building the Block Diagram

Block Diagram Objects ........ 5-1
Block Diagram Terminals ........ 5-1
Control and Indicator Data Types ........ 5-2
Constants ........ 5-2
Block Diagram Nodes ........ 5-3
Functions Overview ........ 5-4
Adding Terminals to Functions ........ 5-4
Built-In VIs and Functions ........ 5-4
Using Wires to Link Block Diagram Objects ........ 5-5
Wire Appearance and Structure ........ 5-5
Wiring Objects ........ 5-5
Selecting Wires ........ 5-6
Correcting Broken Wires ........ 5-6
Block Diagram Data Flow ........ 5-7
Designing the Block Diagram ........ 5-8

## Chapter 6 Running and Debugging VIs


Running VIs ....... 6-1
Correcting Broken VIs ....... 6-2
    Finding Causes for Broken VIs ....... 6-2
    Common Causes of Broken VIs ....... 6-2
Debugging Techniques ....... 6-3
    Execution Highlighting ....... 6-3
    Single-Stepping ....... 6-3
    Probe Tool ....... 6-4
    Breakpoints ....... 6-4
    Error Clusters ....... 6-5


## Chapter 7 Creating VIs and SubVIs


Using Built-In VIs and Functions ....... 7-1
Creating SubVIs ....... 7-1
    Creating an Icon ....... 7-2
    Building the Connector Pane ....... 7-2
    Creating SubVIs from Sections of a VI ....... 7-3
    Designing SubVI Front Panels ....... 7-3
Saving VIs ....... 7-4
Customizing VIs ....... 7-4


## Chapter 8 Loops and Structures


For Loop and While Loop Structures ....... 8-1
    For Loops ....... 8-1
    While Loops ....... 8-2
    Controlling Timing ....... 8-3
    Auto-Indexing Loops ....... 8-4
        Auto-Indexing to Set the For Loop Count ....... 8-4
        Auto-Indexing with While Loops ....... 8-5
    Using Loops to Build Arrays ....... 8-5
    Shift Registers in Loops ....... 8-6
        Initializing Shift Registers ....... 8-7
        Stacked Shift Registers ....... 8-8
    Default Data in Loops ....... 8-8
Case Structures ....... 8-9
    Case Selector Values and Data Types ....... 8-10
    Input and Output Tunnels ....... 8-10

## Chapter 9
## Grouping Data Using Strings, Arrays, and Clusters

Grouping Data with Strings ... 9-1
String Controls ... 9-1
Table Controls ... 9-1
Grouping Data with Arrays and Clusters ... 9-2
Arrays ... 9-2
Restrictions ... 9-2
Indexes ... 9-2
Creating Array Controls, Indicators, and Constants ... 9-3
Array Functions ... 9-3
Clusters ... 9-4
Order of Cluster Elements ... 9-4
Cluster Functions ... 9-5
Creating Cluster Controls, Indicators, and Constants ... 9-5

## Chapter 10
## Formula and MathScript Nodes

Creating Formula Nodes ... 10-1
Creating MathScript Nodes ... 10-3

## Chapter 11
## Local Variables, Global Variables, and Race Conditions

Local Variables ... 11-1
Global Variables ... 11-2
Race Conditions ... 11-2

## Chapter 12
## State Machines

State Diagrams ... 12-1
Using the Standard State Machine VI Template ... 12-2
Modifying the Standard State Machine VI ... 12-4
Designing the Front Panel Window ... 12-4
Arranging the Controls and Indicators on the Block Diagram ... 12-5
Defining the States of the State Machine ... 12-6
Configuring the No Money State ... 12-7
Configuring the Five Cents State ... 12-9
Configuring the Ten Cents State ... 12-11

## Chapter 13
## Developing a Program

Brainstorming ........ 13-1
Identifying Inputs/Outputs ........ 13-2
Identifying Potential Problems ........ 13-2
Developing Flowcharts ........ 13-3
Implementing the Code ........ 13-4
Verifying the Code ........ 13-5
Programming in a Group ........ 13-6
Analyzing the Project ........ 13-6

## Appendix A
## Technical Support and Professional Services

# About This Manual

Use this manual as a tutorial to familiarize yourself with the LabVIEW graphical programming environment and the basic LabVIEW features you can use to build *FIRST* Robotics Competition (FRC) applications.

This manual describes LabVIEW programming concepts, techniques, features, VIs, and functions you can use to create FRC applications. This manual does not include specific information about each palette, tool, menu, dialog box, control or indicator, 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 *Related Documentation* section of Chapter 1, *Introduction to LabVIEW*, for more information about the *LabVIEW Help* and how to access it.

The *LabVIEW Robotics Programming Guide for the FIRST Robotics Competition* provides information about robotics programming concepts and reference information about the *FIRST* Robotics Competition VIs. Refer to the *Related Documentation* section of Chapter 1, *Introduction to LabVIEW*, for more information about the *LabVIEW Robotics Programming Guide for the FIRST Robotics Competition* and how to access it.

## Conventions

This manual uses the following conventions:

» 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 indicators 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. Italic text 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, operations, variables, filenames, and extensions. |
| **`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. |

# 1

# Introduction to LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) 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 the order of program execution, LabVIEW uses dataflow programming, where the flow of data through the nodes on the block diagram determines the execution order of the VIs and functions. VIs, or virtual instruments, are LabVIEW programs that imitate physical instruments.

In LabVIEW, you build a user interface by using 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. This graphical source code is also known as G code or block diagram code. In some ways, the block diagram resembles a flowchart.

Refer to Chapter 4, *Building the Front Panel*, for more information about the front panel. Refer to Chapter 5, *Building the Block Diagram*, for more information about the block diagram.

## LabVIEW VI Templates and Example VIs

Use the LabVIEW VI templates, example VIs, and tools as a starting point to help you design and build VIs.

### LabVIEW VI Templates

The built-in VI templates include the subVIs, functions, structures, and front panel objects you need to get started building common applications. VI templates open as untitled VIs that you must save. Select **File»New** to display the **New** dialog box, which lists the built-in VI templates. You also can display the **New** dialog box by clicking the **New** link in the **Getting Started** window.

## LabVIEW Example VIs

LabVIEW searches among hundreds of example VIs you can use and incorporate into VIs that you create. You can modify an example to fit an application, or you can copy and paste from one or more examples into a VI that you create. Browse or search the example VIs with the NI Example Finder by selecting **Help»Find Examples**.

The FRC software provides example VIs that demonstrate how to use the FRC VIs to build robotics applications. Access these example VIs by navigating to the `National Instruments\LabVIEW 8.6\examples\FRC` directory.

Refer to the NI Developer Zone at `ni.com/zone` for additional example VIs.

# Related Documentation

LabVIEW includes extensive documentation for new and experienced LabVIEW users.

## LabVIEW Help

Use the *LabVIEW Help* to access information about LabVIEW programming concepts, step-by-step instructions for using LabVIEW, and reference information about LabVIEW VIs, functions, palettes, menus, and tools. The *LabVIEW Help* also contains reference information about FRC-specific VIs and dialog boxes.

The *LabVIEW Help* includes links to the technical support resources on the National Instruments Web site, such as NI Developer Zone, the KnowledgeBase, and the Product Manuals Library.

Access the *LabVIEW Help* by selecting **Help»Search the LabVIEW Help**. You also can print a help topic or a book of help topics from the *LabVIEW Help*.

Refer to the *LabVIEW Help* for more information about printing help topics.

## LabVIEW Manuals

The following manuals contain information that you might find helpful as you use LabVIEW:

- *Getting Started with LabVIEW*—Use this manual as a tutorial 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 documentation resources, keyboard shortcuts, data type terminals, and tools for editing, execution, and debugging.

These documents are available as PDFs in the `National Instruments\LabVIEW 8.6\manuals` directory. You must have Adobe Reader 6.0.1 or later installed to view or search 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.

## FRC-Specific Resources

The following resources contain information that you might find helpful as you use LabVIEW to build FRC applications:

- *LabVIEW Robotics Programming Guide for the FIRST Robotics Competition*—Use this manual to access information about robotics, programming concepts, reference information about the *FIRST* Robotics Competition VIs, and guidelines for troubleshooting in LabVIEW. Access this manual by navigating to the `National Instruments\LabVIEW 8.6\manuals` directory and opening `FRC_Programming_Guide.pdf`.
- *cRIO-FRC Operating Instructions and Specifications*—Use this manual to learn about installing, configuring, and using the CompactRIO device for the *FIRST* Robotics Competition. Access this manual by navigating to the `National Instruments\CompactRIO\manuals` directory and opening `crio-frc_Operating_Instructions.pdf`.
- *FRC Community*—Refer to the FRC Community Web site at `http://firstcommunity.usfirst.org/` for official information about the FRC competition, including rules and regulations as well as support information.

# 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 interface to 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.

Click the **Blank VI** link in the **Getting Started** window to create a new, blank VI. You also can create a new, blank VI by pressing the <Ctrl-N> keys.

# Front Panel

The front panel is the user interface of the VI.

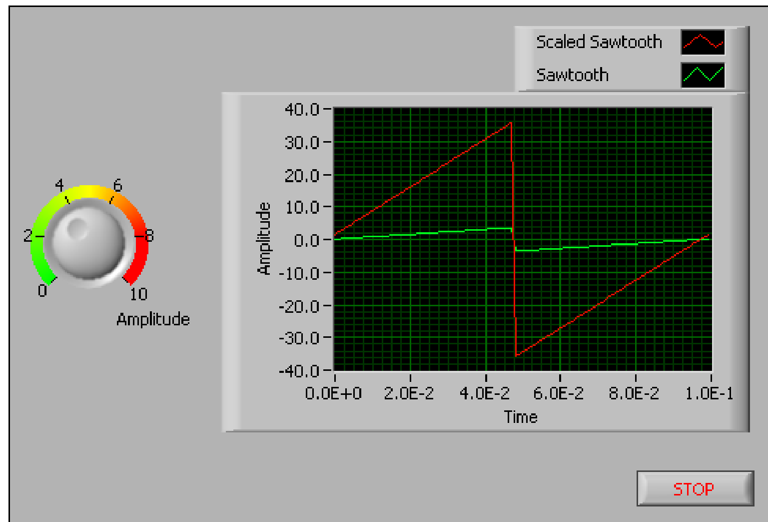



**Figure 2-1.** Front panel of a VI

You build the front panel using controls and indicators, which are the interactive input and output terminals, respectively, of the VI. Controls are knobs, push buttons, dials, and other input mechanisms. Indicators are graphs, LEDs, and other output displays. Controls simulate instrument input mechanisms and supply data to the block diagram of the VI. Indicators simulate instrument output mechanisms 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, also known as G code or block diagram code. Front panel objects appear as terminals on the block diagram.

The following VI contains several primary block diagram objects—terminals, functions, and wires.

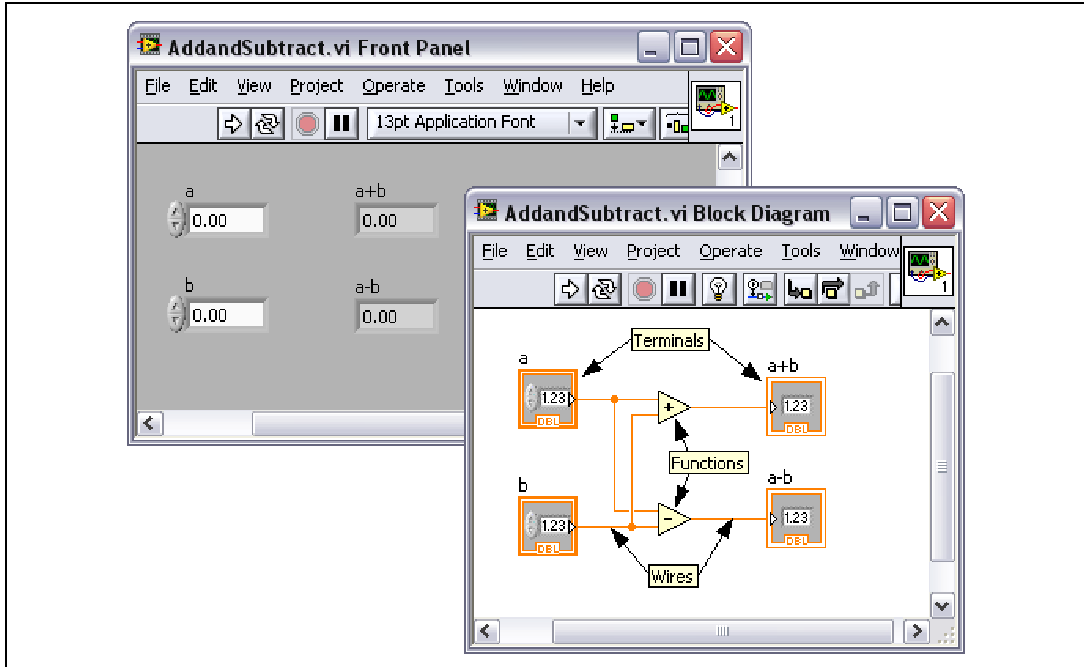



**Figure 2-2.** Block diagram and corresponding front panel

Refer to Chapter 5, *Building the Block Diagram*, for more information about the block diagram.

## Terminals

Terminals represent the data type of controls and indicators. 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.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data you enter into the front panel controls enters the block diagram through the control terminals. Returned data values pass from the block diagram to the front panel through the indicator terminals. In Figure 2-2, **a** and **b** are control terminals, and **a+b** and **a–b** are indicator terminals.

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

## 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 examples of nodes.

Refer to the *Block Diagram Nodes* section of Chapter 5, *Building the Block Diagram*, for more information about nodes.

## Wires

Wires transfer data among block diagram objects. 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 the data source 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. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types.

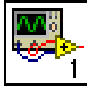
Refer to the *Using Wires to Link Block Diagram Objects* section of Chapter 5, *Building the Block Diagram*, for more information about wires.

## 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.

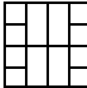
Refer to Chapter 8, *Loops and Structures*, for more information about structures.

# 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. The icon and connector pane correspond to the function prototype in text-based programming languages. 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.

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 a 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.

Refer to the *Building 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 *Creating SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about subVIs.

You also can customize the appearance and behavior of a VI.

Refer to the *Customizing VIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about customizing a VI.

# 3

# LabVIEW Environment

The LabVIEW environment includes the **Getting Started** window, the **Context Help** window, the **Project Explorer** window, and the **Navigation** window. LabVIEW also includes palettes, tools, and menus to build the front panels and block diagrams of VIs. LabVIEW includes three palettes: the **Controls** palette, the **Functions** palette, and the **Tools** palette. You can customize the **Controls** and **Functions** palettes, and you can set several work environment options.

## Getting Started Window

The **Getting Started** window appears when you launch LabVIEW. Use this window to create new VIs, select among the most recently opened LabVIEW files, find examples, and launch the *LabVIEW Help*. You also can access information and resources to help you learn about LabVIEW, such as specific manuals, help topics, and resources on the National Instruments Web site, `ni.com`.

In LabVIEW for FRC, the **Getting Started** window contains links to FRC-specific resources and examples. You also can create an FRC cRIO robotics project or an FRC dashboard project from the **Getting Started** window. Refer to the *LabVIEW Robotics Programming Guide for the FIRST Robotics Competition*, accessible by navigating to the `National Instruments\LabVIEW 8.6\manuals` directory and opening `FRC_Programming_Guide.pdf`, for more information about creating FRC projects and developing a robotics application.

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

# 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, dialog box components, and items in the **Project Explorer** window. You also can use the **Context Help** window to determine exactly where to connect wires to a VI or function.

Refer to the *Using Wires to Link Block Diagram 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, shown at left, on the toolbar.

You also can display the window by pressing the <Ctrl-H> keys.

If a corresponding *LabVIEW Help* topic exists for an object the **Context Help** window describes, a blue **Detailed help** link appears in the **Context Help** window. Also, the **Detailed help** button in the **Context Help** window, shown at left, is enabled.

Click the link or the button to display more information about the object.

# Project Explorer Window

Use the **Project Explorer** window to create and edit LabVIEW projects. Use projects to group together LabVIEW files and non-LabVIEW files, create build specifications, and deploy or download files to targets. Select **File»New Project** to display the **Project Explorer** window.

The **Project Explorer** window includes two pages, the **Items** page and the **Files** page. The **Items** page displays the project items as they exist in the project tree. The **Files** page displays the project items that have a corresponding file on disk. You can organize filenames and folders on this page. Project operations on the **Files** page both reflect and update the contents on disk.

Refer to the *LabVIEW Help* for more information about the **Project Explorer** window.

# Navigation Window

The **Navigation** window displays an overview of the active front panel in edit mode or the active block diagram. Use the **Navigation** window to navigate large front panels or block diagrams. Click an area of the image in the **Navigation** window to display that area in the front panel or block diagram window. You also can click and drag the image in the **Navigation** window to scroll through the front panel or block diagram. Portions of the front panel or block diagram that are not visible appear dimmed in the **Navigation** window.

Select **View»Navigation Window** to display the **Navigation** window. You also can display the window by pressing the <Ctrl-Shift-N> keys.

# 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.

Select **View»Controls Palette** or right-click the front panel workspace to display 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 *LabVIEW Help* for more information about the types of built-in VIs and functions.

Select **View»Functions Palette** or right-click the block diagram workspace to display the **Functions** palette.

# Navigating the Controls and Functions Palettes

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.

Click the black arrows on the left side of the **Controls** or **Functions** palette to expand or collapse subpalettes. These arrows appear only if you set the palette format to **Category (Standard)** or **Category (Icons and Text)**.

Click the **Search** button on the **Controls** or **Functions** palette toolbar to perform text-based searches to locate controls, VIs, or functions on the palettes. While a palette is in search mode, click the **Return** button to exit search mode and return to the palette.

# 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 you select on the palette. Use the tools to operate and modify front panel and block diagram objects.

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. Automatic tool selection is enabled by default.

Select **View»Tools Palette** to display the **Tools** palette.

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

# Menus and Toolbars

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

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 keyboard shortcuts.

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**, selecting **Environment** from the **Category** list, and removing the checkmark from the **Use abridged menus** checkbox.

**Note** Some menu items are unavailable while a VI runs.

## Shortcut Menus

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

## 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 by default. 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.

## VI Toolbar

Use the buttons on the VI toolbar to run VIs, pause VIs, abort VIs, debug VIs, configure fonts, and align, group, and distribute objects.

Refer to Chapter 6, *Running and Debugging VIs*, for more information about some of the toolbar buttons, or refer to the *LabVIEW Help* for a complete list and description of the toolbar buttons.

## Project Explorer Window Toolbars

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

Refer to the *Project Explorer Window* section of this chapter for more information about LabVIEW projects.

# Customizing Your Work Environment

You can use the **Options** dialog box, available by selecting **Tools»Options**, to select a palette format and set other work environment options. 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 **Category** list at the left side of the **Options** dialog box to select among the different categories of options.

# 4

# Building the Front Panel

The front panel is the user interface of a VI. Generally, you design the front panel first and 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 can select controls and indicators from the **Controls** palette and place them on the front panel. Select **View»Controls Palette** to display the **Controls** palette.

## 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 mechanisms. Indicators are graphs, LEDs, and other output displays. Controls simulate instrument input mechanisms and supply data to the block diagram of the VI. Indicators simulate instrument output mechanisms and display data the block diagram acquires or generates. The most common controls and indicators are numeric, Boolean, and string controls and indicators.

### Numeric Controls and Indicators

Use numeric controls and indicators to enter and display numeric data. You can resize these front panel objects horizontally to accommodate more digits. Change the value of a numeric control in any of the following ways:

- Use the Operating tool or the 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 numeric control.
- Use the Operating tool or the Labeling tool to place the cursor to the right of the digit you want to change and press the up or down arrow keys.

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 by right-clicking the numeric object and selecting **Format and Precision** from the shortcut menu to display the **Format and Precision** page of the **Numeric Properties** dialog box.

## Boolean Controls and Indicators

Use the Boolean controls and indicators located on the **Boolean** and **Classic Boolean** palettes to create 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 exceeds a certain level.

Boolean controls have six types of mechanical action that allow you to customize the behavior of Boolean objects. Use Boolean controls to create front panels that resemble the behavior of physical instruments. Use the shortcut menu to customize the appearance and behavior of Boolean objects.

## String Controls and Indicators

Use string controls or indicators to manipulate and display text. Use the Operating or Labeling tool to enter or edit text in a string control on the front panel. By default, new or changed text does not pass to the block diagram until you terminate the edit session. At run time, 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.

Right-click a string control or indicator to select a display type for the text in the control or indicator, such as password display or hex display.

Refer to the *Grouping Data with Strings* section of Chapter 9, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about string display types.

# Configuring Front Panel Objects

Use **Properties** dialog boxes or shortcut menus to configure how controls and indicators appear or behave on the front panel. Use **Properties** dialog boxes when you want to set several properties of an object at once. Use shortcut menus to configure common control and indicator properties. The options available in **Properties** dialog boxes and shortcut menus differ for different front panel objects. Any option you set using a shortcut menu is reflected in the **Properties** dialog box, and any option you set using the **Properties** dialog box is reflected in the shortcut menu.

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

## 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 place a toggle switch on the front panel, it appears as a control because a toggle switch is usually an input mechanism. If you place an LED on the front panel, it appears 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 numeric control and a numeric indicator because you can have a numeric input or a numeric output.

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. 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. If you

replace a numeric terminal with another numeric terminal, LabVIEW tries to preserve the original representation. However, if the control does not support the new data type, the new object retains its own data type. Wires from the terminal of the object 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 select **Edit»Copy** and **Edit»Paste** to copy objects to the clipboard and paste them 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 changing the color of front panel objects, aligning and distributing front panel objects, and so on.

## Coloring Objects

You can change the color of most front panel objects and the front panel and block diagram workspaces. You cannot change the color of system controls and indicators because these objects appear in the colors you have set up for your system.

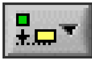
Use the Coloring tool to right-click an object or workspace to change the color of front panel objects or of the front panel and block diagram workspaces. You also can change the default colors for some objects by selecting **Tools»Options** and selecting **Colors** from the **Category** list.

Color can distract the user from important information so use color logically, sparingly, and consistently, if at all.

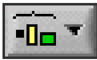
## Aligning and Distributing Objects

Use grid alignment to align objects to the front panel grid when you place, move, or resize them. Select **Edit»Enable Panel Grid Alignment** to enable grid alignment on the front panel. Select **Edit»Disable Panel Grid Alignment** to disable grid alignment and use the visible grid to align objects manually. You also can press the <Ctrl-#> keys to enable or disable the grid alignment.

Select **Tools»Options** and select **Alignment Grid** from the **Category** list to hide or customize the grid.



To align objects after you place them, select the objects and select the **Align Objects** pull-down menu, shown at left, on the toolbar or select **Edit»Align Items**.



To space objects evenly, select the objects and select the **Distribute Objects** pull-down menu, shown at left, on the toolbar or select **Edit» Distribute Items**.

You also can use grid alignment on the block diagram.

## Grouping and Locking Objects

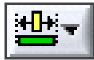
Grouped objects maintain their relative arrangement and size when you use the Positioning tool to move and resize them. Locked objects maintain their location on the front panel, and you cannot delete them until you unlock them. Use the Positioning tool to select the front panel objects you want to group and lock together. Click the **Reorder** button, shown at left, on the toolbar and select **Group or Lock** from the pull-down menu. 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.



To resize multiple objects to the same size, select the objects and select the **Resize Objects** pull-down menu, shown at left, 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.

## 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.

A rectangle marked by a dashed border defines where space will be inserted. Release the mouse button and the <Ctrl> key to add the space.

# 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 can display a separate owned label called a unit label for numeric controls and indicators by right-clicking the numeric control or indicator and selecting **Visible Items»Unit Label** from the shortcut menu.

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.

# 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 identify what actions to perform. You can design front panels that look similar to instruments or other devices.

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.

# 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, also known as G code or block diagram code.

## Block Diagram Objects

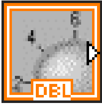
Objects on the block diagram include terminals and nodes. You build block diagrams by connecting the objects with wires. The color and symbol of each terminal indicate the data type of the corresponding control or indicator. Constants are terminals on the block diagram that supply fixed data values to the block diagram.

### 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 values you enter into the front panel controls enter the block diagram through the control terminals. During execution, the output data values flow to the indicator terminals, where they exit the block diagram, reenter the front panel, and appear in front panel indicators.

LabVIEW has control and indicator terminals, node terminals, constants, and specialized terminals on structures. 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.



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 control 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.

Right-click a terminal and remove the checkmark next to the **View As Icon** shortcut menu item 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.

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.

## Control and Indicator Data Types

Common control and indicator data types include floating-point numeric, integer numeric, time stamp, enumerated, Boolean, string, array, cluster, path, dynamic, waveform, refnum, and I/O name. Refer to the *LabVIEW Help* for the complete list of control and indicator data types with their symbols and uses.

The color and symbol of each terminal indicate the data type of the corresponding control or indicator. Many data types have a corresponding set of functions that can manipulate the data, such as the String functions on the **String** palette that correspond to the string data type.

## 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.

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

Create a user-defined constant by right-clicking an input terminal of a VI or function and selecting **Create»Constant** from the shortcut menu. 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 operators, functions, or statements.
- **SubVIs**—VIs used on the block diagram of another VI, comparable to subroutines.

  Refer to the *Creating SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about using subVIs on the block diagram.
- **Express VIs**—SubVIs designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.
- **Structures**—Execution control elements, such as For Loops, While Loops, Case structures, Flat and Stacked Sequence structures, Timed structures, and Event structures.

  Refer to Chapter 8, *Loops and Structures*, for more information about using structures.

Refer to the *LabVIEW Help* for the complete list of block diagram 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 or edit a function.

## 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 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 functions, but you cannot remove a terminal that is already wired. You must first delete the existing wire to remove the terminal.

Refer to the *Using Wires to Link Block Diagram Objects* section of this chapter for more information about wiring objects.

## Built-In VIs and Functions

The **Functions** palette also includes the VIs that ship with LabVIEW. Use these VIs and functions as subVIs in an application to reduce development time. Click the **View** button on the **Functions** palette and select **Always Visible Categories»Show All Categories** from the shortcut menu to display all categories on the **Functions** palette.

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 and functions.

Refer to the *LabVIEW Help* for detailed information about all built-in VIs and functions.

# 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 the data source to many VIs and functions that read the data, similar to passing required parameters in text-based programming languages. You must wire all required block diagram terminals. Otherwise, the VI is broken and cannot 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 *Correcting Broken VIs* section of Chapter 6, *Running and Debugging VIs*, for more information about broken VIs.

## Wire Appearance and Structure

Wires are different colors, styles, and thicknesses depending on their data types, similar to how the color and symbol of a terminal indicate the data type of the corresponding control or indicator.

Refer to the *Control and Indicator Data Types* section of this chapter for more information about data types. Refer to the *Block Diagram Data Flow* section of this chapter for more information about data flow.

## 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.

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.

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.

## 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 wire 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 to display 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.

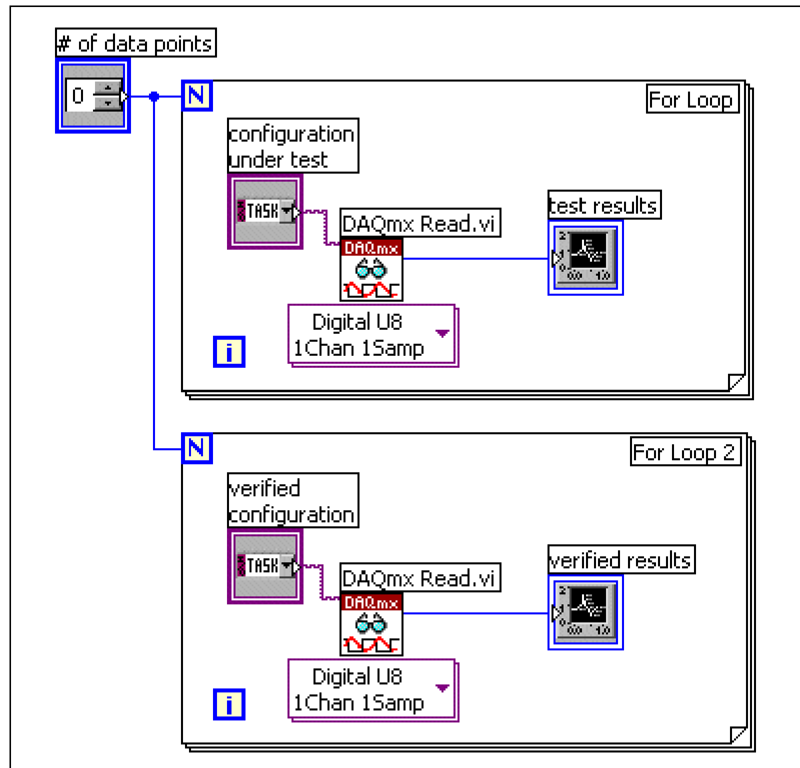
You can remove all broken wires by selecting **Edit»Remove Broken Wires** or by pressing the <Ctrl-B> keys.

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

# Block Diagram Data Flow

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

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




In the preceding figure, each For Loop can execute when it receives all required inputs. The only required input for each For Loop is the value of the count terminal, which the **# of data points** control specifies. Therefore, when the **# of data points** control passes a value to the For Loops, both For Loops can execute simultaneously.

# 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 works 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 *Creating 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 Clusters* section of Chapter 6, *Running and Debugging VIs*, for more information about handling errors.
- 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. Placing a terminal or icon on top of a wire gives the appearance that a connection exists when it does not.
- Use free labels to document code on the block diagram.

  Refer to the *Labeling* section of Chapter 4, *Building the Front Panel*, for more information about using free labels.

# 6

# Running and Debugging VIs

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

## 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.

Refer to the *Building the Connector Pane* section of Chapter 7, *Creating VIs and SubVIs*, for more information about creating connector panes.

A VI runs when you click the **Run** or **Run Continuously** buttons or the single-stepping buttons on the block diagram toolbar. While the VI runs, the **Run** button changes to a darkened arrow, shown at left, to indicate that the VI is running.

You cannot edit a VI while the VI runs.

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.

# Correcting Broken VIs

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

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

## Finding Causes for Broken VIs

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

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

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

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.

## Common Causes of Broken VIs

The following list contains common reasons why a VI might be broken:

- The block diagram contains a broken wire because of a mismatch of data types or a loose, unconnected end.

  Refer to the *Correcting Broken Wires* section of Chapter 5, *Building the Block Diagram*, for information about correcting broken wires.

- A required block diagram terminal is unwired.

  Refer to the *Using Wires to Link Block Diagram Objects* section of Chapter 5, *Building the Block Diagram*, for information about setting required inputs and outputs.

- A subVI is broken or you edited its connector pane after you placed its icon on the block diagram of the VI.

  Refer to the *Creating SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for information about subVIs.

# Debugging Techniques

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

## 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 values move from node to node through a VI.

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

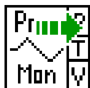
During execution highlighting, if the **error out** cluster reports an error, the error value appears next to **error out** with a red border. If no error occurs, OK appears next to **error out** with a green border.

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

**Note** You must run a VI in order to see data pass through a probe in the VI.

## 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 passes 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 **Operate» Breakpoints** and clicking the **Find** button.

## Error Clusters

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

VIs and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs.

Error handling in LabVIEW follows the dataflow model. Just as data values 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.

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

Some VIs, functions, and structures that accept Boolean data also recognize an error cluster. For example, you can wire an error cluster to the Boolean inputs of the Select, Quit LabVIEW, or Stop functions. If an error occurs, the error cluster passes a TRUE value to the function.

Refer to the *Clusters* section of Chapter 9, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about clusters.

# 7 Creating VIs and SubVIs

A VI can serve as a user interface or as an operation you use frequently. After you learn how to build a front panel and block diagram, you can create your own VIs and subVIs and customize these VIs.

## Using Built-In VIs and Functions

LabVIEW includes built-in VIs and functions to help you build specific applications, such as data acquisition VIs and functions, VIs that access other VIs, VIs that communicate with other applications, and so on. You can use these VIs as subVIs in an application to reduce development time. Before you build a new VI, consider searching the **Functions** palette for similar VIs and functions and using an existing VI as the starting point for the new VI.

## Creating SubVIs

After you build a VI, you can use it in another VI. A VI called from the block diagram of another VI is called a subVI. To create a subVI, you need to build a connector pane and create an icon.

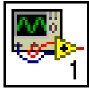
A subVI node corresponds to a subroutine call in text-based programming languages. The node is not the subVI itself, just as a subroutine call statement in a program is not the subroutine itself. A block diagram that contains several identical subVI nodes calls the same subVI several times.

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 or text 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.

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.

To create a subVI, you need to create an icon and build a connector pane.

# Creating an Icon



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.

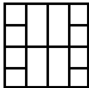
An icon is a graphical representation of a VI. It can contain text, images, or a combination of both. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI.

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.

Refer to the National Instruments Web site at `ni.com/info` and enter the info code `expnr7` for standard graphics to use in a VI icon.

# Building the Connector Pane



To use a VI as a subVI, you need to build a connector pane, shown at left.

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

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 and select **Show Connector** from the shortcut menu to display the connector pane. The connector pane appears in place of the icon.

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

Each rectangle on the connector pane represents a terminal. Use the rectangles to assign inputs and outputs. The default connector pane pattern is $4 \times 2 \times 2 \times 4$. If you anticipate changes to the VI that require a new input or output, keep the default connector pane pattern to leave extra terminals unassigned.

You can assign up to 28 terminals to a connector pane. If the 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 9, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about grouping data using clusters.

## 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 selecting **Edit» Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.

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 SubVI Front Panels

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 on the front panel as they appear in the connector pane. Place the controls on the left of the front panel and the indicators on the right. Place any **error in** clusters on the lower left of the front panel and any **error out** clusters on the lower right.

Refer to the *Building the Connector Pane* section of this chapter for more information about setting up a connector pane.

# Saving VIs

Select **File»Save** to save a VI. When you save a VI, use a descriptive name so you can identify the VI later. Descriptive names, such as `Temperature Monitor.vi` and `Serial Write & Read.vi`, make a VI easy to identify. 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 `#`.

**Note** If you have several VIs of the same name saved on your computer, carefully organize the VIs in different directories or LLBs to avoid LabVIEW referencing the wrong subVI when running the top-level VI.

# 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.

Select **File»VI Properties** to configure the appearance and behavior of a VI. Use the **Category** pull-down menu at the top of the **VI Properties** dialog box to select from several different option categories.

The **VI Properties** dialog box includes the following option categories:

- **General**—Use this page to determine 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 for the VI.
- **Documentation**—Use this page to add a description of the VI and link to a help file topic.
- **Security**—Use this page to lock or password-protect a VI.
- **Window Appearance**—Use this page to customize the window appearance of VIs, such as the window title and style.
- **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.

- **Editor Options**—Use this page to set the size of the alignment grid for the current VI and to change the style of the 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.

# 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.

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

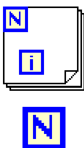
- **For Loop**—Executes a subdiagram a set number of times.
- **While Loop**—Executes a subdiagram until a condition occurs.
- **Case structure**—Contains multiple subdiagrams, only one of which executes depending on the input value passed to the structure.

Right-click the border of a structure to display its 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, specifies 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.

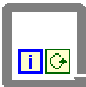
The iteration count always starts at zero. During the first iteration, the iteration terminal returns `0`.

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 output contains 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 in Loops* section of this chapter for more information about adding shift registers to a loop.

## While Loops



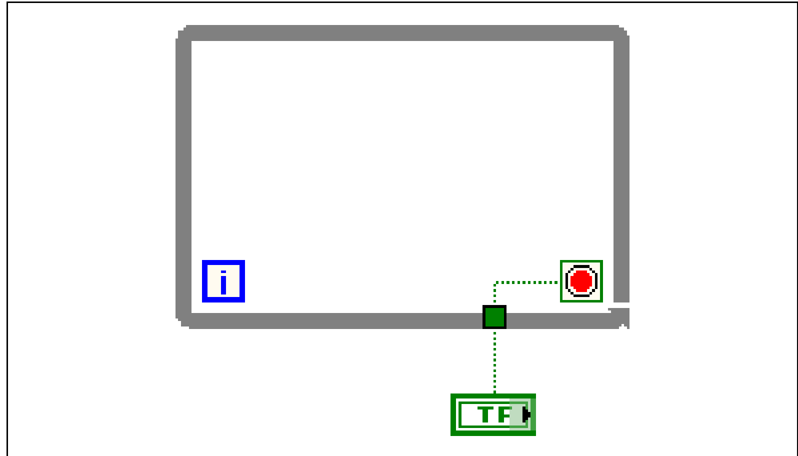
Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown at left, executes a subdiagram until a condition occurs.

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.

If the conditional terminal is **Stop if True**, you place the corresponding Boolean control outside the While Loop, and you set the Boolean control to FALSE, you cause an infinite loop, as shown in the following figure.




You also cause an infinite loop if the Boolean control outside the loop is set to TRUE and the conditional terminal is **Continue if True**. Changing the value of the Boolean control does not stop the infinite loop because the value is read only once before the loop starts. To stop an infinite loop, you must abort the VI by clicking the **Abort Execution** button on the toolbar.

The iteration terminal (an output terminal) of a While Loop, 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 in Loops* section of this chapter for more information about adding shift registers to a loop.

## Controlling Timing

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

## 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 the *Arrays* section of Chapter 9, *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. 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.

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.

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.

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 lesser 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. As another example, 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.

### 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 occurs. 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 *Arrays* section of Chapter 9, *Grouping Data Using Strings, Arrays, and Clusters*, for more information about arrays.

Refer to the `National Instruments\LabVIEW 8.6\examples\general\arrays.llb` for examples of building arrays.

## Shift Registers in Loops

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

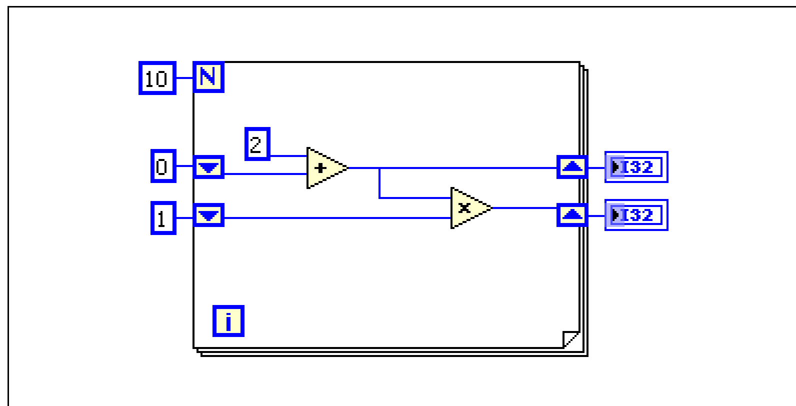
Use shift registers when you want to pass values from previous iterations through the loop to the next iteration. A shift register appears as a pair of terminals, shown at left, directly opposite each other on the vertical sides of the loop border.

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

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

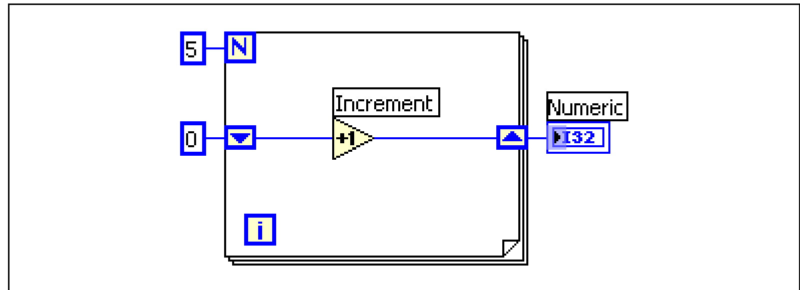
A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type.

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



## Initializing Shift Registers

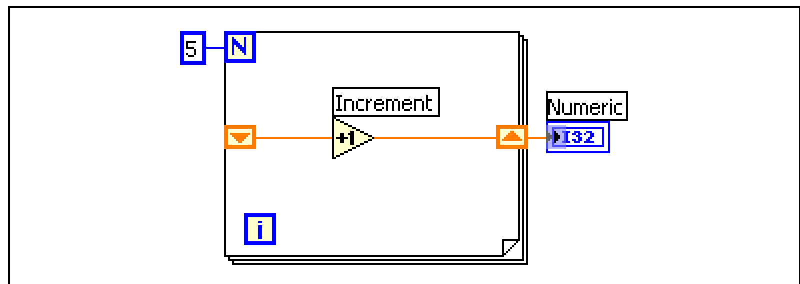
Initializing a shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop, as shown in the following figure.




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

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

Use an uninitialized shift register to preserve state information between subsequent executions of a VI. The following figure shows an uninitialized shift register.



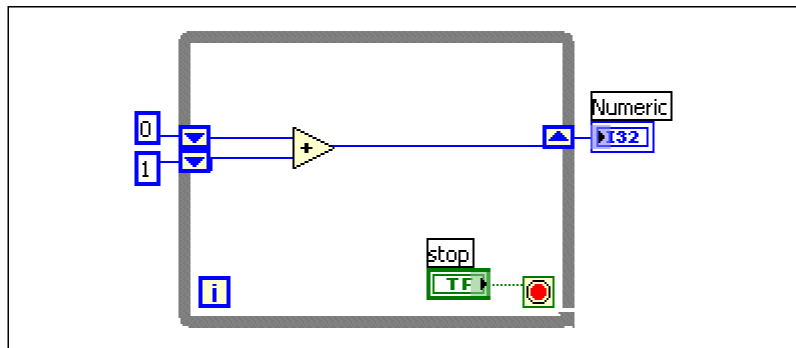

In the preceding figure, the For Loop executes five times, incrementing the value the shift register carries by one each time. The first time you run the VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer. After five iterations of the For Loop, the shift register passes

the final value, 5, to the indicator, and the VI quits. The next time you run the VI, the shift register begins with a value of 5, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value, 10, to the indicator. If you run the VI again, the shift register begins with a value of 10, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI.

## Stacked Shift Registers

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

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




In the preceding block diagram, values from previous iterations pass to the next iteration, with the most recent iteration value stored in the top-left shift register. The bottom shift register stores the second-most-recent iteration value.

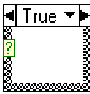
# Default Data in Loops

While Loops produce default data when the shift register is not initialized.

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.

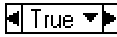
Refer to the *LabVIEW Quick Reference Card*, available by navigating to the `National Instruments\LabVIEW 8.6\manuals` directory and opening `LV_Quick_Reference.pdf`, for more information about default values for data types.

# 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 switch 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 unspecified 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`.

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 numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest integer. If you type a floating-point value in the case selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

## Input and Output Tunnels

You can create multiple input and output tunnels for a Case structure. Inputs are available to all cases, but cases do not have 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 even 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.

# 9

# 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.

## Grouping Data with 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 numeric values
- Storing numeric data to disk
- Instructing or prompting the user with dialog boxes

On the front panel, strings appear as tables, text entry boxes, and labels. LabVIEW includes built-in VIs and functions you can use to edit, format, and parse strings.

### String Controls

Use 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.

### Table Controls

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.

Refer to the *Arrays* section of this chapter for more information about arrays.

# 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.

### Restrictions

You cannot create arrays of arrays. However, you can use a multidimensional array or create an array of clusters where each cluster contains one or more arrays. Also, you cannot create an array of subpanel controls, tab controls, .NET controls, ActiveX controls, charts, or multiplot XY graphs.

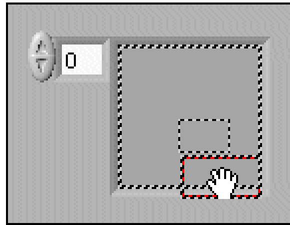
Refer to the *Clusters* section of this chapter for more information about clusters.

### Indexes

Locating 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.

## 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 the following figure, 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.




The array shell automatically resizes to accommodate the new object.

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

## Array Functions

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

- Extracting individual data elements from an array
- Inserting, deleting, or replacing data elements in an array
- Splitting arrays

Use the Build Array function to build an array programmatically. You also can use a loop to build an array.

Refer to the *Using Loops to Build Arrays* section of Chapter 8, *Loops and Structures*, for information about using loops to build arrays.

Refer to the *LabVIEW Style Checklist* in the *LabVIEW Help* for more information about minimizing memory usage when using Array functions in a loop.

## Clusters

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

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

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

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

### Order of Cluster Elements

Although cluster and array elements are both ordered, you must unbundle all cluster elements at once or 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.

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 to each other, 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 value in one cluster corresponds in cluster order to a string in another cluster, the wire on the block diagram appears broken and the VI does not run. If the numeric values are different representations, LabVIEW coerces them to the same representation.

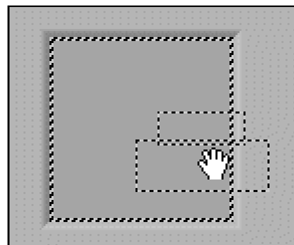
## Cluster Functions

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

- Extracting individual data elements from a cluster
- Adding individual data elements to a cluster
- Breaking a cluster out into its individual data elements

## Creating Cluster Controls, Indicators, and Constants

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



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

# 10

# Formula and MathScript 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.

The MathScript Node also is a text-based node you can use to perform mathematical operations on the block diagram. However, the MathScript Node can execute LabVIEW MathScripts and `.m` files.

## Creating Formula Nodes

Complete the following steps to create a Formula Node.

1. Place a Formula Node on the block diagram.
2. Use the Labeling tool or the Operating tool to enter the equations you want to calculate inside the Formula Node. Each assignment must have only a single variable on the left side of the assignment (=). Each assignment must end with a semicolon (;). Confirm that you are using the correct Formula Node syntax.

   If a syntax error occurs, click the broken **Run** button to display the **Error list** window. LabVIEW marks the syntax error with a # symbol.

**Tip** Add comments to the text in a Formula Node by enclosing them inside a slash-asterisk pair (`/*comment*/`), or after a double-slash (`//comment`).

3. Create an input terminal for each input variable by right-clicking the Formula Node border and selecting **Add Input** from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the Labeling tool or the Operating tool, except when the VI is running.

   Variable terminals are case sensitive. There is no limit to the number of terminals or equations in a Formula Node. You can change a terminal type or remove a terminal.

4. Create an output terminal for each output variable by right-clicking the Formula Node border and selecting **Add Output** from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the Labeling tool or the Operating tool, except when the VI is running. Output variables have thicker borders than input variables.

**Note** No two inputs and no two outputs can have the same name. However, an output can have the same name as an input.

5. (Optional) The default data type for output terminals is double-precision, floating-point numeric. To change the data type, create an input terminal with exactly the same name as the output terminal and wire a data type to that input terminal. Doing so also provides a default value for the terminal. You also can use the Formula Node syntax to define the variable inside the Formula Node. For example, `int32 y;` changes the data type of the output terminal **y** to 32-bit integer.

6. Wire the input and output terminals of the Formula Node to their corresponding terminals on the block diagram. All input terminals must be wired. Output terminals do not have to be wired.

Refer to the *LabVIEW Help* for more information about the Formula Node.

# Creating MathScript Nodes

Complete the following steps to create and run a VI that uses a LabVIEW MathScript.

1. Place a MathScript Node on the block diagram.
2. Use the Operating or Labeling tool to enter the following script in the MathScript Node:

   ```
   a = rand(50, 1)

   plot(a)
   ```

3. Add an output to the MathScript Node and create an indicator for the output.
   a. Right-click the right side of the MathScript Node frame and select **Add Output** from the shortcut menu.
   b. Enter `a` in the output terminal to add an output for the *a* variable in the MathScript.
   c. Change the data type of the output terminal. In MathScript, the default data type for any new input or output is a **Scalar»DBL**. Right-click the **a** output and select **Choose Data Type»Matrix» Real Matrix** from the shortcut menu.
   d. Right-click the **a** output terminal and select **Create»Indicator** from the shortcut menu to create a matrix indicator labeled **a**.
4. Right-click the **error out** output terminal and select **Create»Indicator** from the shortcut menu to create an **error out** indicator.
5. Run the VI. LabVIEW invokes the MathScript server, creates a vector of random values, plots that information to a graph, and displays the values that make up the vector in the **Real Matrix** front panel indicator.

Refer to the *LabVIEW Help* for more information about the MathScript Node.

# 11

# Local Variables, Global Variables, and Race Conditions

In LabVIEW, you read data from or write data to a front panel object using its block diagram terminal. However, a front panel object has only one block diagram terminal, and your application might need to access the data in that terminal from more than one location.

Local and global variables pass information between locations in the application that you cannot connect with a wire. Use local variables to access front panel objects from more than one location in a single VI. Use global variables to access and pass data among several VIs.

## 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 nodes 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.

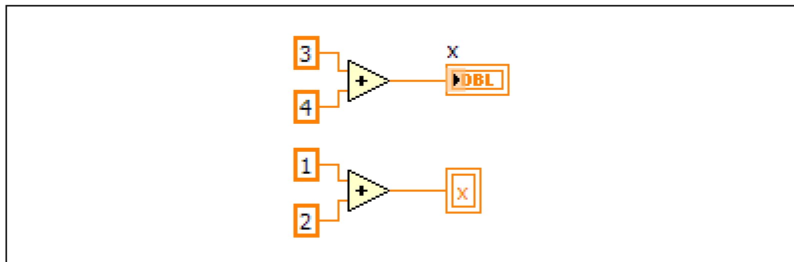
# Global Variables

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.

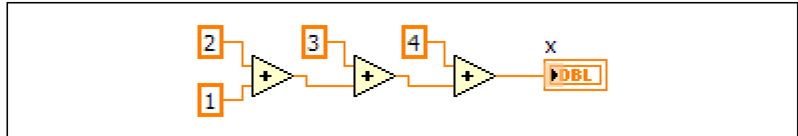
For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You must use a global variable to terminate both loops with a single Boolean control. If both loops were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

# 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. Because the outcome of the VI depends on which action executes on the shared resource first, race conditions cause unpredictable outcomes. Race conditions often occur with the use of local and global variables or an external file, although race conditions can exist any time more than one action updates the value of the same stored data. The following block diagram shows an example of a race condition with a local variable.



The output of this VI, the value of local variable x, depends on which operation runs first. Because each operation writes a different value to x, you cannot determine whether the outcome will be 7 or 3. In some programming languages, a top-down dataflow paradigm ensures execution order. In LabVIEW, you can use wiring to perform multiple operations on a variable while avoiding race conditions. The following block diagram performs addition operations using wiring instead of a local variable.



**Tip** If you must perform more than one action on a local or global variable, make sure you determine the order of execution.

Race conditions also occur when two operations try to update a global variable in parallel. In order to update the global variable, an operation reads the value, modifies it, and writes it back to the location. When the first operation performs the read-modify-write action and the second operation follows after, the outcome is correct and predictable. When the first operation reads, and then the second operation reads, both operations modify and write a value. This action causes the read-modify-write race condition and produces invalid or missing values.

You can avoid race conditions associated with global variables by using functional global variables. Functional global variables are VIs that use loops with uninitialized shift registers to hold global data. A functional global variable usually has an **action** input parameter that specifies which task the VI performs. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation. Using one functional global variable instead of multiple local or global variables ensures that only one operation executes at a time, so you never perform conflicting operations or assign conflicting values to stored data.

# 12

# State Machines

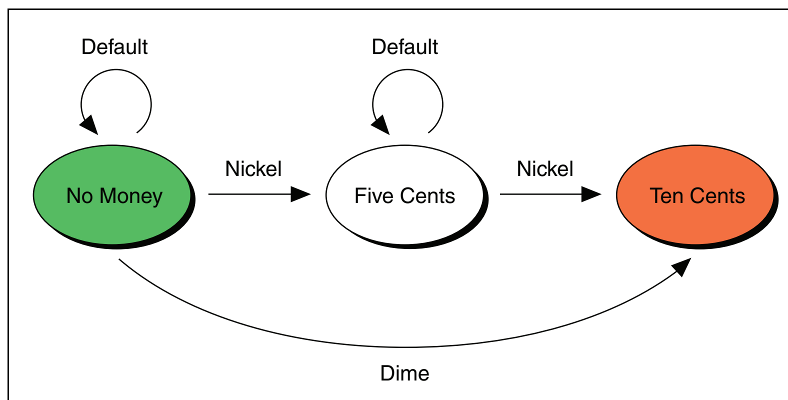
Use the state machine architecture to implement complex decision-making algorithms represented by state diagrams or flow charts. A state machine consists of multiple states, each of which executes code and determines the next code to which to transition. The state machine can have an initial state and a terminal state, as well as one or more intermediate states.

## State Diagrams

You can use a state diagram to represent the states and transitions of a state machine graphically. To create an effective state diagram, you must know the various states of the application and how they relate to one another. By visualizing the various execution states of the application, you improve the overall design of the application.

For example, consider a vending machine that sells candy for 10 cents. The vending machine can have the following states: No Money, Five Cents, and Ten Cents. The No Money state is the initial state. In the No Money state, the vending machine continues to wait for money to be inserted. In the Five Cents state, the vending machine contains five cents and continues to wait for additional money to be inserted. The Ten Cents state is the terminal state. In the terminal state, the vending machine returns the candy.

To transition between the initial state and the second state of the vending machine, you must insert a nickel. To transition between the second state and the terminal state, you must insert another nickel or a dime. You also can transition directly from the initial state to the terminal state by inserting a dime. The following state diagram describes this behavior.



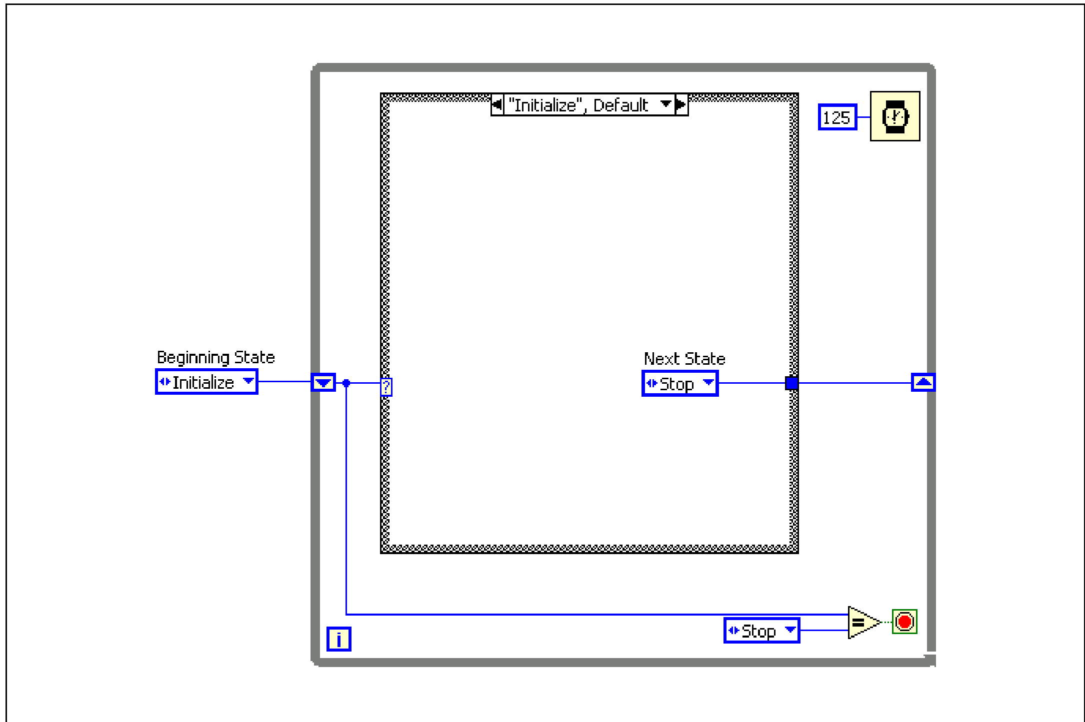

This state diagram can help you visualize how to design the actual state machine.

# Using the Standard State Machine VI Template

You can use a VI to represent the state machine of the vending machine. You can create the VI from scratch, or you can use a VI template that LabVIEW provides.

Complete the following steps to create a VI using the Standard State Machine VI template.

1. Click the **New** link in the **Getting Started** window or select **File»New** to display the **New** dialog box.
2. From the **Create New** list, navigate to **VI»From Template» Frameworks»Design Patterns»Standard State Machine**.
3. Click the **OK** button.
4. Select **Window»Show Block Diagram** or press the <Ctrl-E> keys to display the block diagram. The VI looks similar to the following figure.




5. Save the VI as `Vending Machine.vi` in an easily accessible location.

Notice that this VI consists of a While Loop and a Case structure, as well as an enum constant that specifies the current state. In this template, only two states are available: Initialize and Stop. The Case structure determines the code that each state executes. The While Loop executes until the Stop state is reached.

If you run this VI without any modifications, the state machine begins in the Initialize state. The While Loop passes this state value to the Case structure, and the Initialize case of the Case structure executes. The only code in the Initialize case sets the next state to Stop. The Case structure passes this state value to the shift register on the right border of the While Loop, which in turn passes the value back to the beginning of the next iteration of the While Loop. Because the state value now is Stop, the While Loop stops.

# Modifying the Standard State Machine VI

You can modify the Vending Machine VI to behave according to the state diagram you outlined in the *State Diagrams* section of this chapter.

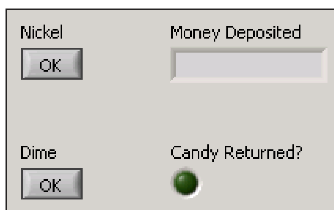
## Designing the Front Panel Window

Complete the following steps to design the front panel window.

1. Press the <Ctrl-E> keys to display the front panel window.
2. Place an OK Button, located on the **Modern»Boolean** palette, on the front panel window.

**Note** Front panel objects appear as terminals on the block diagram. By default, these terminals appear as icon terminals. To conserve space on the block diagram, right-click a terminal and remove the checkmark next to the **View As Icon** shortcut menu item to display the data type for the terminal. You can configure LabVIEW to display terminals for new front panel objects you create as data types by default by selecting **Tools»Options** to display the **Options** dialog box, clicking **Block Diagram** in the **Category** list, and removing the checkmark from the **Place front panel terminals as icons** checkbox.

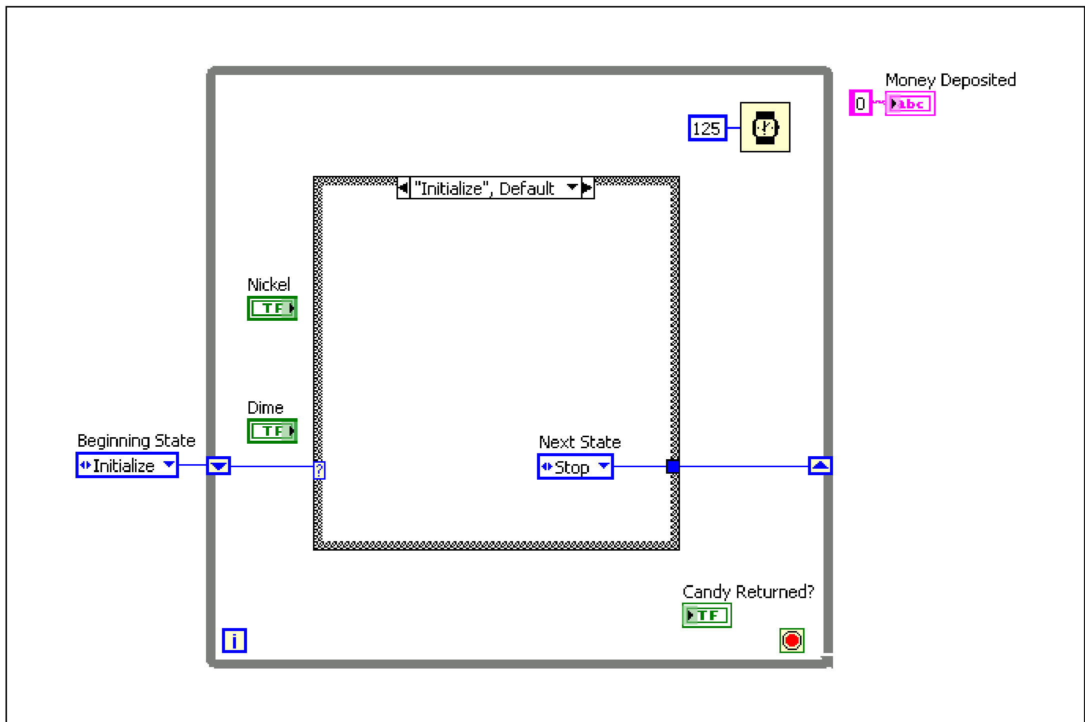
3. Triple-click the **OK Button** label above the OK Button and enter `Nickel` to change the label of the control.
4. Repeat steps 2 and 3 to create a Dime button.
5. Place a String Indicator, located on the **Modern»String & Path** palette, on the front panel window and label it `Money Deposited`.
6. Place a Round LED, located on the **Modern»Boolean** palette, on the front panel window and label it `Candy Returned?`.
7. Arrange the controls and indicators on the front panel similar to the following figure.



## Arranging the Controls and Indicators on the Block Diagram

Complete the following steps to arrange the controls and indicators on the block diagram.

1. Press the <Ctrl-E> keys to display the block diagram.
2. Move the **Nickel** and **Dime** controls to the left of the Case structure but inside the While Loop.
3. Move the **Money Deposited** indicator to the right of the While Loop.
4. Move the **Candy Returned?** indicator inside the While Loop near the conditional terminal.
5. Right-click the **Money Deposited** indicator and select **Create»Constant** from the shortcut menu.
6. Enter `0` in the **Money Deposited** constant to initialize the value of the **Money Deposited** indicator to 0.
7. Delete both the **Equal?** function wired to the conditional terminal of the While Loop and the enum constant wired to the **Equal?** function.
8. Press the <Ctrl-B> keys to delete all broken wires. The block diagram should look similar to the following figure.



## Defining the States of the State Machine

Complete the following steps to define the states of the state machine and configure the Case structure to handle each state in a separate case.

1. Right-click the **Beginning State** enum constant and select **Open Type Def.** from the shortcut menu to display a **Control Editor** window.
2. Right-click the **States** enum control and select **Edit Items** from the shortcut menu to display the **Enum Properties** dialog box.
3. Modify the **Items** list to contain the following enumerated values:

| Items | Digital Display |
|---|---|
| No Money | 0 |
| Five Cents | 1 |
| Ten Cents | 2 |

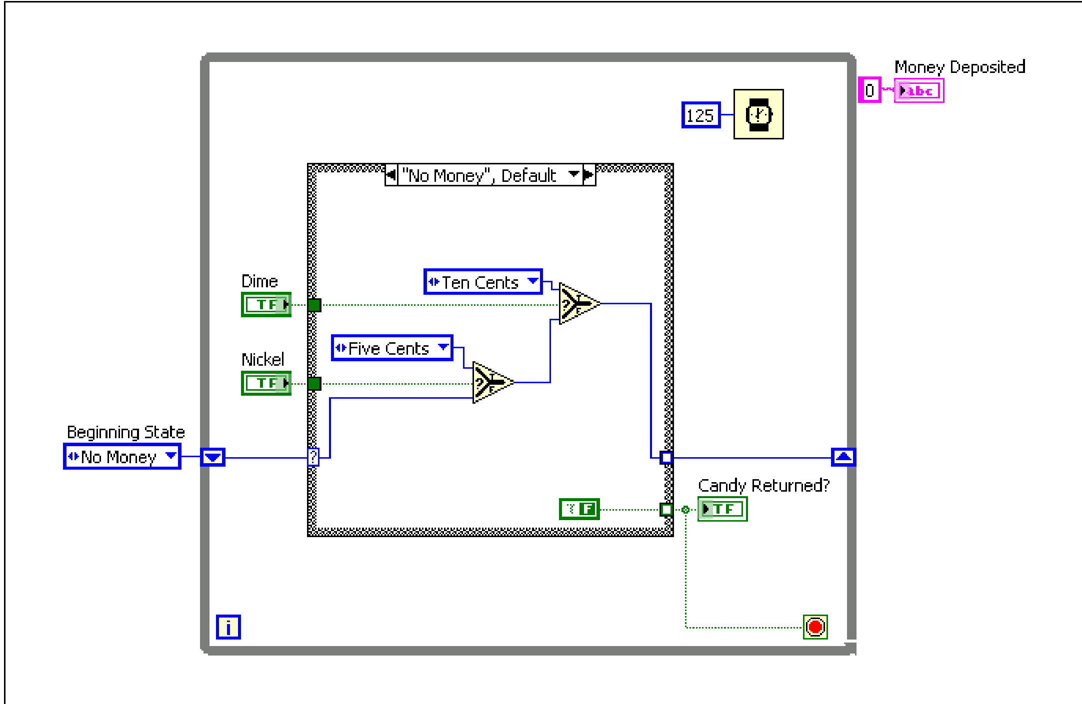
4. Click the **OK** button to return to the **Control Editor** window.
5. Save the control as `Vending States.ctl` in an easily accessible location and close the **Control Editor** window. Notice that the enum constants on the block diagram of the Vending Machine VI update to use the states you defined.
6. Right-click the case selector label at the top of the case structure and select **Add Case for Every Value** from the shortcut menu. You now can configure a case of the Case structure for each of the states of the vending machine.

## Configuring the No Money State

Complete the following steps to configure the No Money state.

1. Click the increment or decrement arrow of the selector label of the Case structure to switch to the No Money case.
2. Place a Select function, located on the **Programming»Comparison** palette, on the block diagram inside the No Money case.
3. Wire the **Nickel** control to the **s** input of the Select function.
4. Place a Vending States constant, accessible by clicking **Select a VI** on the **Functions** palette and navigating to the Vending States control you saved, on the block diagram to the left of the Select function.
5. Select **Five Cents** from the drop-down list of the Vending States constant.
6. Wire the Vending States constant to the **t** input of the Select function.
7. Wire the selector terminal of the Case structure to the **f** input of the Select function. The Select function returns the Five Cents state, if the Nickel control is TRUE, or the current state, if the Nickel control is FALSE.
8. Repeat steps 2 through 6 using the Dime control and a Ten Cents state.
9. Wire the **s? t:f** output of the first Select function to the **f** input of the second Select function. The second Select function returns the Ten Cents state, if the Dime control is TRUE, or the state corresponding to the result of the first Select function, if the Dime control is FALSE.
10. Delete the **Next State** enum control and the wire connecting it to the enum output tunnel of the Case structure.
11. Wire the **s? t:f** output of the second Select function to the enum output tunnel of the Case structure. The output state of the No Money case passes to the shift register on the right border of the While Loop, which in turn passes the value back to the beginning of the next iteration of the While Loop.
12. Place a False Constant, located on the **Programming»Boolean** palette, on the block diagram inside the No Money case.
13. Wire the **False Constant** to both the conditional terminal of the While Loop and to the **Candy Returned?** indicator.

The No Money case should look similar to the following figure.



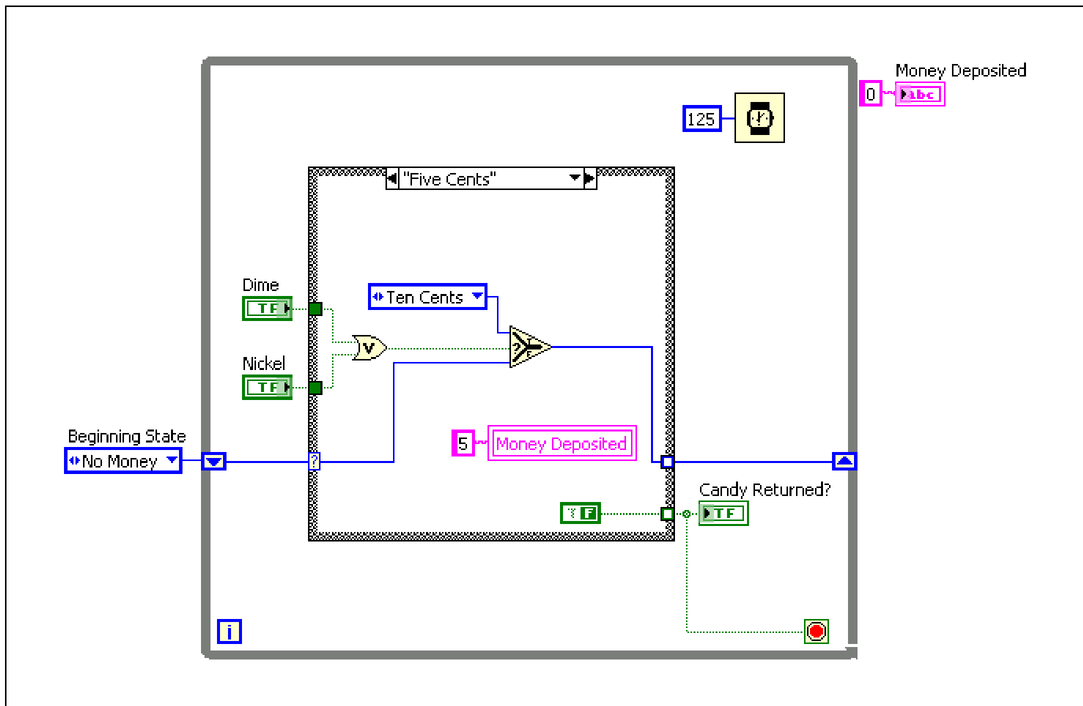
## Configuring the Five Cents State

Complete the following steps to configure the Five Cents state.

1. Click the increment or decrement arrow of the selector label of the Case structure to switch to the Five Cents case.
2. Place an Or function, located on the **Programming»Boolean** palette, on the block diagram inside the Five Cents case.
3. Wire the **Dime** control to the **x** input of the Or function.
4. Wire the **Nickel** control to the **y** input of the Or function.
5. Place a Select function on the block diagram to the right of the Or function.
6. Wire the **x .or. y?** output of the Or function to the **s** input of the Select function.
7. Place a Vending States control on the block diagram between the Or function and the Select function.
8. Select **Ten Cents** from the drop-down list of the Vending States constant.
9. Wire the Vending States constant to the **t** input of the Select function.
10. Wire the selector terminal of the Case structure to the **f** input of the Select function. The Select function returns the Ten Cents state, if either the Nickel control or the Dime control is TRUE, or the current state, if neither the Nickel control nor the Dime control are TRUE.
11. Delete the **Next State** enum control and the wire connecting it to the enum output tunnel of the Case structure.
12. Wire the **s? t:f** output of the Select function to the enum output tunnel of the Case structure. The output state of the Five Cents case passes to the shift register on the right border of the While Loop, which in turn passes the value back to the beginning of the next iteration of the While Loop.
13. Place a False Constant inside the Five Cents case.
14. Wire the **False Constant** to the Boolean output tunnel of the Case structure.
15. Right-click the **Money Deposited** indicator, select **Create»Local Variable** from the shortcut menu, and place the local variable in the Five Cents case of the Case structure.
16. Right-click the **Money Deposited** local variable and select **Create» Constant** from the shortcut menu.

17. Enter `5` in the Money Deposited constant. When the vending machine is in the Five Cents case, the **Money Deposited** indicator displays a value of **`5`**.

The Five Cents case should look similar to the following figure.

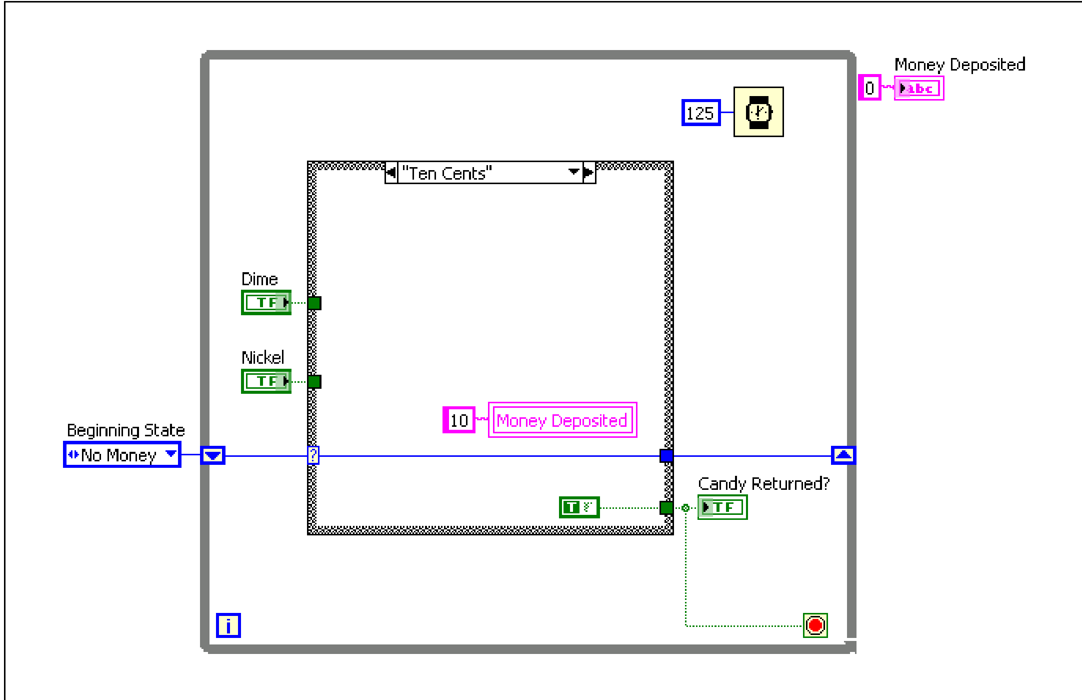


## Configuring the Ten Cents State

Complete the following steps to configure the Ten Cents state.

1. Click the increment or decrement arrow of the selector label of the Case structure to switch to the Ten Cents case.
2. Wire the selector terminal of the Case structure to the enum output tunnel. After the vending machine reaches the Ten Cents state, the While Loop stops, and the state no longer changes. Therefore, the current state passes directly through the Ten Cents case.
3. Place a True Constant, located on the **Programming»Boolean** palette, inside the Ten Cents case.
4. Wire the **True Constant** to the Boolean output tunnel of the Case structure. In the Ten Cents state, the **True Constant** passes a value of TRUE to the **Candy Returned?** indicator and to the conditional terminal of the While Loop. Because the value of the conditional terminal is **Stop if True**, the While Loop then stops.
5. Right-click the **Money Deposited** indicator, select **Create»Local Variable** from the shortcut menu, and place the local variable in the Ten Cents case of the Case structure.
6. Right-click the **Money Deposited** local variable and select **Create» Constant** from the shortcut menu.
7. Enter `10` in the Money Deposited constant. When the vending machine is in the Ten Cents case, the **Money Deposited** indicator displays a value of **`10`**.

The Ten Cents case should look similar to the following figure.




You now can run the VI and observe the values of the **Money Deposited** and **Candy Returned?** indicators on the front panel when you click the **Nickel** and **Dime** buttons.

# 13

# Developing a Program

The basic features of LabVIEW programming were covered in previous chapters. You can use these features to develop a program in LabVIEW. Before developing a program, you must plan accordingly. Program development often includes the following stages: brainstorming, developing flowcharts, implementing the code, and verifying the code.

## Brainstorming

Start a project by brainstorming. Consider the following questions during brainstorming:

- What do you want to accomplish with the program?
- What do you want the outcomes of the program to be? What actions must the program perform in order to return the outcomes you want?
- What resources do you need to operate the program correctly? Can you think of any potential problems that might disrupt the execution of the program?

Write down ideas for the program during brainstorming so others can see the thoughts and ideas. If a project involves multiple participants, group brainstorming sessions allow participants to share thoughts.

Refer to the *Programming in a Group* section of this chapter for more information about programming in a group.

During brainstorming, write down everything that comes to mind, no matter how unfeasible an idea seems. When you consider ideas for a project, you can establish a clearer understanding of the program.

Consider, for example, a program for selling train tickets. The program must account for several factors such as the prices of different train tickets, types of discounts, methods of payment, and train schedules.

During brainstorming, you might consider actions of the ticket program to include selling tickets, providing train schedules, selling different classes of service, accepting credit cards, calculating change, and selling refreshments. Selling refreshments might be an unlikely action of the ticket

program. However, you can excuse unnecessary ideas later during development.

When you list actions the program might perform, you also can identify possible inputs and outputs of the program.

## Identifying Inputs/Outputs

Every program has inputs and outputs. Inputs include all elements the program uses to make calculations and process data to produce the end results, or outputs. Without inputs and outputs, the program has no functionality.

Inputs of the ticket program might include train destinations, ticket types, discount types, and currencies. The purpose of a program is to manipulate the inputs you enter and return output values. Therefore, outputs of the ticket program might be the ticket information, ticket type, and change amount.

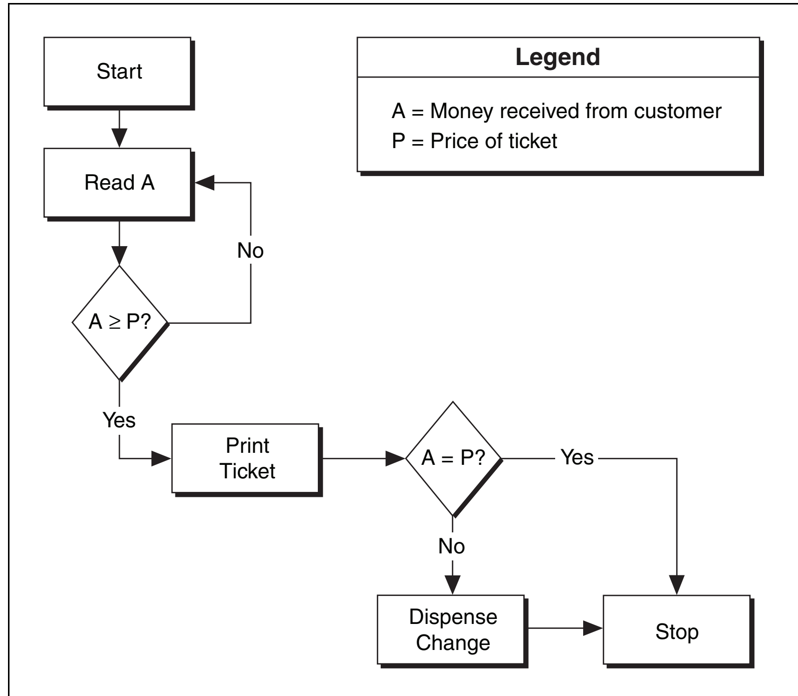
By considering possible inputs and outputs, you gain an understanding of the program before actual programming begins.

## Identifying Potential Problems

By identifying potential problems before program development, you can reduce delays in programming and implementation. In the ticket program, some problems might include train schedule updates due to delays, incorrect money amounts, and limited train capacities.

# Developing Flowcharts

After brainstorming, map out a program by developing a flowchart. Flowcharts illustrate the program steps from start to finish. For the ticket program example, the following figure shows a flowchart for when a customer purchases a ticket.




The rectangular symbols in the figure represent actions. These actions include starting the program, reading input values, returning output values, and stopping the program. An example of reading input values is reading the amount of money a customer pays. An example of returning output values is displaying change amounts. Flowchart action symbols can have a maximum of one exiting arrow because each symbol must represent a single action with a single output.

Always include the start and stop actions when you develop flowcharts. These symbols represent when a program starts and stops execution.

The diamond symbols in the flowchart represent decisions. Decisions determine the flow of a flowchart. Use decisions to check conditions based on input. The next path of the execution depends on whether the specified

condition is met. In the preceding flowchart, when the program reaches the first decision, the program checks whether the money the customer pays is greater than or equal to the ticket price. If the customer does not pay enough money, the program returns to the *Read A* action. If the customer pays enough money and the condition is met, the ticket prints and the program continues.

By developing flowcharts, you can illustrate the flow of a program before writing code. You also can determine whether certain inputs are appropriate for a program, and whether they produce the outputs you map out.

# Implementing the Code

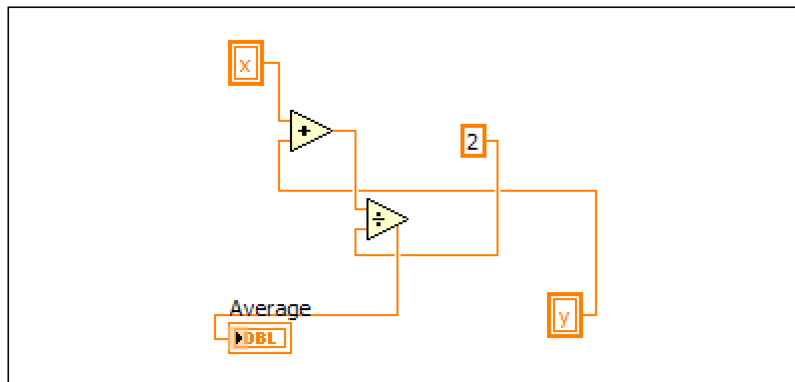
After brainstorming and developing a flowchart for a program, you can start writing code. Use the resources you created during brainstorming as references during this stage.

During implementation, use the following programming practices:

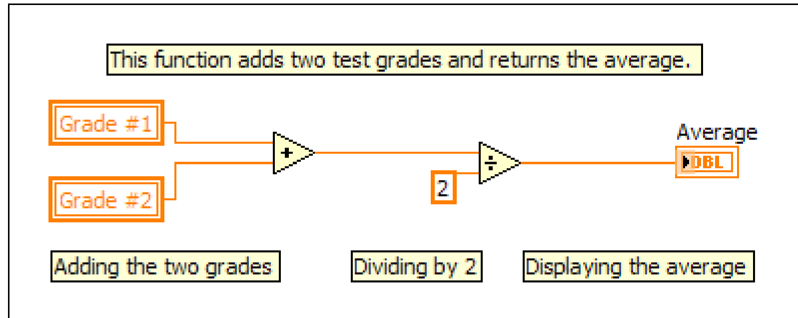
- Thoroughly document all code.
- Give controls and indicators relevant names.
- Make code spacing readable and clean.

Provide thorough documentation with the code so other programmers can view the code and understand the task you are trying to implement. Thorough documentation saves time when others work with the code.

Also make sure labels describe the behavior of controls and indicators. Consider the following block diagram.



The block diagram is difficult to read, and the purpose of the program is unclear. The following figure shows the same program in a readable and linear style.




Detailed documentation clarifies the purpose of the program. The code flows left-to-right, and spacing makes the block diagram readable. By documenting all code, giving controls and indicators relevant names, and making code spacing readable and clean, you reduce confusion when others read the program you write.

# Verifying the Code

In the last stage of program development, you must verify the code. You must complete extensive testing to make sure the program is error- and bug-free. Define and execute tests to account for as many use cases as possible. Also implement and test the error-handling that checks for invalid inputs.

You can use the following scenarios to test the train ticket program:

- Buy a ticket with insufficient money.
- Purchase a ticket for a train that has left.
- Pay more money than the amount due.

Testing is necessary to ensure the reliability of a program. For example, consider an automated toll road machine that accepts only exact change. Toll road machines must process and calculate a variety of coins in a brief amount of time. If a machine does not process the coins correctly, drivers might receive tickets mistakenly. Testing the processing of different coin combinations can reduce calculation errors. Similarly, you can test different use cases of a program you develop to identify and correct bugs before you implement the program.

# Programming in a Group

Complex projects require the contribution of multiple team members. If multiple developers work on the same project, define programming responsibilities, interfaces, and coding standards at the beginning of the project. If you develop a program with a group, consider the following practices:

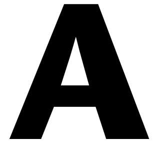
- Split up portions of code such that everyone can program efficiently.
- Select the team integrator. The integrator is responsible for combining all code written by the team into the final program. The integrator must communicate with the programmers to ensure all code can work together effectively.
- Ensure each workload is realistic. Certain portions of the project require more work than others. For example, the integrator typically has a larger workload.
- Set realistic deadlines and communicate them clearly to the team. By planning specific deadlines for project milestones, you can measure progress more effectively.

# Analyzing the Project

At the end of the development process, consider having a post-project analysis meeting to discuss what went well and what did not. Each developer must evaluate the project honestly and discuss obstacles that reduced the quality level of the project. Consider the following questions during a post-project analysis meeting:

- What are we doing right? What works well?
- What are we doing wrong? What can we improve?
- Do any specific areas of the design or code need work?
- Are the quality systems working? Can we catch more problems if we change the quality requirements? Can we find better ways to get the same results?

Similar analysis meetings at major milestones help the team to correct problems mid-schedule instead of waiting until the end of the release cycle.



# Technical Support and Professional Services

Visit the following sections of the award-winning National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. 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, National Instruments Alliance Partner 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.