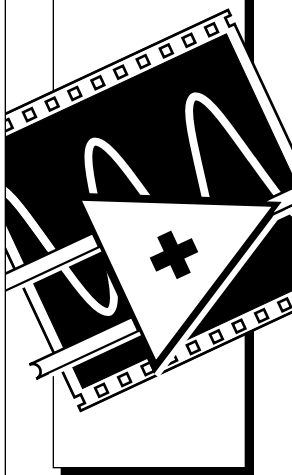


LabVIEW



LabVIEW[®] Tutorial Manual

January 1996 Edition
Part Number 320998A-01



Internet Support

GPiB: gpib.support@natinst.com

DAQ: daq.support@natinst.com

VXI: vxi.support@natinst.com

LabVIEW: lv.support@natinst.com

LabWindows: lw.support@natinst.com

HiQ: hiq.support@natinst.com

E-mail: info@natinst.com

FTP Site: [ftp.natinst.com](ftp://ftp.natinst.com)

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



Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077

BBS United Kingdom: 01635 551422

BBS France: 1 48 65 15 59



FaxBack Support

(512) 418-1111 or (800) 329-7177



Telephone Support (U.S.)

Tel: (512) 795-8248

Fax: (512) 794-5678 or (800) 328-2203



International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,

Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,

Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,

Italy 02 48301892, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,

Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,

Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

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 manual 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 THEREFORE 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.

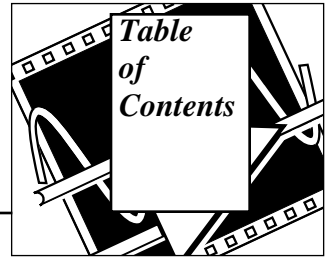
Trademarks

LabVIEW® and NI-488M™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.



About This Manual

Organization of This Manual	ix
Conventions Used in This Manual	xi
Related Documentation	xii
Customer Communication	xiii
Customer Education	xiii

Chapter 1 Introduction to LabVIEW

Chapter Information	1-2
What Is LabVIEW?	1-2
How Does LabVIEW Work?	1-3
Installing LabVIEW	1-4
LabVIEW Files	1-4
Virtual Instruments	1-4
Front Panel	1-5
Front Panel Toolbar	1-7
Block Diagram	1-9
Hierarchy	1-11
Icon/Connector Pane	1-12
Tools Palette	1-14
Editing Techniques	1-15
Controls Palette	1-18
Controls and Indicators	1-19
Numeric Controls and Indicators	1-19
Boolean Controls and Indicators	1-20
Configuring Controls and Indicators	1-20
Functions Palette	1-21
Building a VI	1-21
Front Panel	1-23
Block Diagram	1-24
Wiring Techniques	1-27
Tip Strips	1-28
Showing Terminals	1-28
Wire Stretching	1-29

Selecting and Deleting Wires	1-29
Bad Wires	1-30
Create & Wire Controls, Constants, and Indicators	1-30
Run the VI	1-31
Documenting the VI	1-32
Saving and Loading VIs	1-34
Summary	1-36

Chapter 2

Creating a SubVI

Understanding Hierarchy.....	2-1
Creating the SubVI	2-1
Icon	2-2
Icon Editor Tools and Buttons	2-2
Connector	2-4
Using a VI as a SubVI	2-6
Front Panel	2-6
Block Diagram	2-7
Block Diagram Toolbar	2-8
Some Debugging Techniques	2-9
Opening, Operating, and Changing SubVIs	2-12
Hierarchy Window	2-13
Search Hierarchy	2-14
Online Help for SubVI Nodes	2-15
Simple/Complex Help View	2-15
Links to Online Help Files	2-16
Summary	2-17

Chapter 3

Loops and Charts

Using While Loops and Charts	3-1
Front Panel	3-2
Block Diagram	3-3
Mechanical Action of Boolean Switches	3-6
Adding Timing	3-7
For Loop	3-9
Numeric Conversion	3-10
Using a For Loop	3-12
Front Panel	3-12
Block Diagram	3-13
Shift Registers	3-14
Using Shift Registers	3-16

Front Panel	3-16
Block Diagram	3-17
Multiplot Charts	3-19
Customizing Charts	3-20
Different Chart Modes	3-22
Summary	3-23
Additional Topics	3-24
Customizing Charts	3-24
Faster Chart Updates	3-24
Stacked Versus Overlaid Plots	3-24
Using Loops	3-24
Testing a While Loop before Execution	3-24
Using Uninitialized Shift Registers	3-26

Chapter 4

Arrays, Clusters, and Graphs

Arrays	4-1
Array Controls, Constants, and Indicators	4-1
Graphs	4-2
Creating an Array with Auto-Indexing	4-2
Front Panel	4-2
Block Diagram	4-4
Multiplot Graphs	4-7
Polymorphism	4-8
Using Auto-Indexing on Input Arrays	4-9
Using Auto-Indexing to Set the For Loop Count	4-10
Using the Initialize Array Function	4-11
Using the Graph and Analysis VIs	4-12
Front Panel	4-13
Block Diagram	4-13
Using Arrays	4-15
Creating and Initializing Arrays	4-15
Using the Build Array Function	4-16
Finding the Size of an Array	4-18
Using the Array Subset Function	4-18
Using the Index Array Function	4-19
Summary	4-22
Additional Topics	4-23
More About Arrays	4-23
Efficient Memory Usage: Minimizing Data Copies	4-23
Customizing Graphs	4-23
Graph Cursors	4-24

Intensity Plots	4-25
Data Acquisition Arrays (Windows, Macintosh, and Sun)	4-25
Graph Examples	4-25

Chapter 5

Case and Sequence Structures and the Formula Node

Using the Case Structure	5-1
Front Panel	5-1
Block Diagram	5-2
VI Logic	5-4
Using the Sequence Structure	5-5
Front Panel	5-5
Modifying the Numeric Format	5-5
Setting the Data Range	5-7
Block Diagram	5-8
Formula Node	5-11
Using the Formula Node	5-13
Front Panel	5-14
Block Diagram	5-15
Summary	5-16
Additional Topics	5-17
More Information on Case and Sequence Structures	5-17
Timing with Sequence Structures	5-17
More Information on Formula Nodes	5-17
Artificial Data Dependency	5-17

Chapter 6

Strings and File I/O

Strings	6-1
Creating String Controls and Indicators	6-1
Strings and File I/O	6-2
Using String Functions	6-2
Front Panel	6-2
Block Diagram	6-3
Using Format Strings	6-4
Front Panel	6-4
Block Diagram	6-5
More String Functions	6-7
Front Panel	6-7
Block Diagram	6-8
File I/O	6-9
File I/O Functions	6-10

Writing to a Spreadsheet File	6-11
Front Panel	6-12
Block Diagram	6-12
Appending Data to a File	6-14
Front Panel	6-14
Block Diagram	6-15
Reading Data from a File	6-16
Front Panel	6-17
Block Diagram	6-17
Using the File I/O Functions	6-18
Specifying a File	6-18
Paths and Refnums	6-19
File I/O Examples	6-20
Summary	6-20
Additional Topics	6-21
Datalog Files	6-21
Binary Byte Stream Files	6-22
Error I/O in File I/O Functions	6-22

Chapter 7

Customizing VIs

VI Setup	7-1
Setting Window Options	7-2
SubVI Node Setup	7-3
Using Setup Options for a SubVI	7-3
Front Panel	7-4
Block Diagram	7-4
Front Panel	7-7
Block Diagram	7-8
Custom Controls and Indicators	7-10
Summary	7-13
Additional Topics	7-13
Simulating a Control/Indicator	7-13
Using the Control Editor	7-14

Chapter 8

Data Acquisition and Instrument Control

Using LabVIEW to Acquire Data	8-1
About Plug-in Data Acquisition Boards (Windows, Macintosh, and Sun)	8-2
About VISA	8-2
About GPIB	8-3
About Serial Ports	8-4

Using Serial Ports	8-5
Front Panel	8-5
Block Diagram	8-6
About VXI for Windows, Macintosh, and Sun	8-7
About Instrument Drivers	8-8
Using Instrument Drivers	8-9
Front Panel	8-9
Block Diagram	8-10
Using a Frequency Response Test VI	8-13
Front Panel	8-14
Block Diagram	8-15
Writing a Test Sequencer	8-17
Front Panel	8-17
Block Diagram	8-18
Summary	8-19
Additional Topics	8-20
Error Handling	8-20
Waveform Transfers	8-21
ASCII Waveforms	8-21
Binary Waveforms	8-22

Chapter 9 Programming Tips and Debugging Techniques

Programming Tips	9-1
Debugging Techniques	9-5
Finding Errors	9-5
Single Stepping Through a VI	9-5
Execution Highlighting	9-6
Debugging a VI	9-6
Front Panel	9-6
Block Diagram	9-7
Opening the Front Panels of SubVIs	9-9
Summary	9-10

Chapter 10 Program Design

Use Top-Down Design	10-1
Make a List of User Requirements	10-1
Design the VI hierarchy	10-2
Write the Program	10-3
Plan Ahead with Connector Patterns	10-3
SubVIs with Required Inputs	10-5

Good Diagram Style	10-5
Avoid Oversized Diagrams	10-5
Watch for Common Operations	10-6
Use Left-to-Right Layouts	10-7
Check for Errors	10-7
Watch Out for Missing Dependencies	10-9
Avoid Overuse of Sequence Structures	10-10
Study the Examples	10-10

Chapter 11

Where to Go from Here

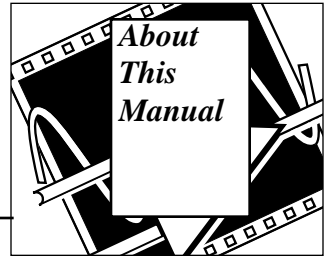
Other Useful Documentation	11-1
For Information on Advanced Topics	11-2

Appendix A

Customer Communication

Glossary

Index



The *LabVIEW Tutorial* contains the information you need to get started with the Laboratory Virtual Instrument Engineering Workbench (LabVIEW) software package. LabVIEW simplifies scientific computation, process control, and test and measurement applications, and you can also use it for a wide variety of other programming applications.

This manual gives you an overview of the fundamental concepts of LabVIEW, and includes lessons to teach you what you need to know to build your own virtual instruments (VIs) as quickly as possible. This manual does not explain every LabVIEW feature. Its goal is to introduce the most important LabVIEW features in the context of addressing programming tasks.

This manual presumes that you know how to operate your computer and that you are familiar with its operating system.

Organization of This Manual

Each chapter discusses a different LabVIEW concept, although you can design a VI that may incorporate several of these basic concepts. Therefore, we encourage you to work through the entire tutorial before you begin building your application.

Some of the chapters in this tutorial include an *Additional Topics* section, which gives an overview of advanced LabVIEW features and refers you to other documentation and example VIs.

This manual is organized as follows:

- Chapter 1, *Introduction to LabVIEW*, describes what LabVIEW is, what a Virtual Instrument (VI) is, how to use the LabVIEW environment (windows, menus, palettes, and tools), how to operate VIs, how to edit VIs, and how to create VIs.

- Chapter 2, *Creating a SubVI*, describes what a subVI is, teaches you how to create the icon and connector, and teaches you how to use a VI as a subVI.
- Chapter 3, *Loops and Charts*, introduces While Loops, teaches you how to display data in a chart, teaches you about shift registers and how to use them, and teaches you how to use For Loops.
- Chapter 4, *Arrays, Clusters, and Graphs*, discusses how to create arrays, use basic array functions, clusters, and graphs. You also learn what polymorphism is, and how to use graphs to display data.
- Chapter 5, *Case and Sequence Structures and the Formula Node*, describes how to use the Case structure and Sequence structure, sequence locals and Formula Nodes.
- Chapter 6, *Strings and File I/O*, teaches you how to create string controls and indicators and teaches you how to use string functions, file input and output operations, save data to files in spreadsheets, and write data to and read data from text files.
- Chapter 7, *Customizing VIs*, shows you how to use the VI and subVI setup options and how to make custom controls and indicators.
- Chapter 8, *Data Acquisition (for Windows, Macintosh, and Sun) and Instrument Control*, discusses how to acquire data from a plug-in data acquisition board, teaches you about VISA, teaches you about GPIB, shows you how to control a serial port interface from LabVIEW, discusses VXI (for Windows, Macintosh, and Sun), teaches you about instrument drivers and how to use them, and teaches you about using a Frequency Response Test VI.
- Chapter 9, *Programming Tips and Debugging Techniques*, gives you tips for programming and debugging VIs and teaches you editing techniques.
- Chapter 10, *Program Design*, offers some techniques to use when creating programs and offers programming style suggestions.
- Chapter 11, *Where to Go From Here*, contains information on other useful resources to examine as you build your LabVIEW applications.
- The Appendix, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.

- The *Glossary* contains an alphabetical list of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this tutorial, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes menus, menu items, or dialog box buttons or options. In addition, bold text denotes VI input and output parameters.
<i>italic</i>	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
<i>bold italic</i>	Bold italic text denotes a note, caution, or warning.
monospace	Monospace font denotes text or characters that you enter using the keyboard. Sections of code, programming examples, syntax examples, and messages and responses that the computer automatically prints to the screen also appear in this font.
<i>italic monospace</i>	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <Shift>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Shift-Delete> .
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence File»Page Setup»Options»Substitute Fonts directs you to pull down the File menu, select the Page Setup item, select Options , and finally select the Substitute Fonts option from the last dialog box.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in <code>drivename\dir1name\dir2name\myfile</code> .

IEEE 488.1 and IEEE 488.2 refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.



Warning: *This icon to the left of bold italicized text denotes a warning, which alerts you to the possibility of damage to you or your equipment.*



Caution: *This icon to the left of bold italicized text denotes a caution, which alerts you to the possibility of data loss or a system crash.*



Note: *This icon to the left of bold italicized text denotes a note, which alerts you to important information.*

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *LabVIEW Analysis VI Reference Manual*
- *LabVIEW Code Interface Reference Manual*
- *LabVIEW Communication VI Reference Manual*
- *LabVIEW Data Acquisition Basics Manual (Windows, Macintosh, and Sun)*
- *LabVIEW Data Acquisition VI Reference Manual (Windows, Macintosh, and Sun)*
- *LabVIEW Instrument I/O VI Reference Manual*
- *LabVIEW User Manual*
- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocols, and Common Commands*
- *LabVIEW Function Reference Manual* available online; printed version available by request.

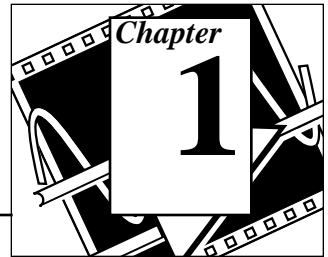
Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in the Appendix, *Customer Communication*, at the end of this manual.

Customer Education

National Instruments offers hands-on LabVIEW Basics and Advanced courses to help you quickly master LabVIEW and develop successful applications. The comprehensive Basics course not only teaches you LabVIEW fundamentals, but also gives you hands-on experience developing data acquisition and instrument control applications. The follow-up Advanced course teaches you how to maximize the performance and efficiency of LabVIEW applications. Contact National Instruments for a detailed course catalog and for course fees and dates.

Introduction to LabVIEW



This chapter describes what LabVIEW is, what a Virtual Instrument (VI) is, how to use the LabVIEW environment (windows, menus, palettes, and tools), how to operate VIs, how to edit VIs, and how to create VIs.

Because LabVIEW is such a feature-rich program development system, this tutorial cannot practically show you how to solve every possible programming problem. Instead, this tutorial explains the theory behind LabVIEW, contains exercises to teach you to use the LabVIEW programming tools, and guides you through practical uses of LabVIEW features as applied to actual programming tasks.

If you would like more training after using this manual, National Instruments offers hands-on LabVIEW courses to help you quickly master LabVIEW and develop successful applications.

The comprehensive LabVIEW Basics course not only teaches you LabVIEW fundamentals, but also gives you hands-on experience developing data acquisition (for Windows, Macintosh, and Sun) and instrument control applications. The follow-up LabVIEW Advanced course teaches you how to maximize the performance and efficiency of LabVIEW applications in addition to teaching you the advanced features of LabVIEW.

For a detailed course catalog and for course fees and dates, refer to the address page on the inside front cover of this manual for information about contacting National Instruments.

Chapter Information

Each chapter begins with a section like the one that follows, listing the learning objectives for that chapter.

You Will Learn:

- What LabVIEW is.
- What a Virtual Instrument (VI) is.
- How to use the LabVIEW environment (windows and palettes).
- How to operate VIs.
- How to edit VIs.
- How to create VIs.

What Is LabVIEW?

LabVIEW is a program development application, much like various commercial C or BASIC development systems, or National Instruments LabWindows. However, LabVIEW is different from those applications in one important respect. Other programming systems use *text-based* languages to create lines of code, while LabVIEW uses a *graphical* programming language, *G*, to create programs in block diagram form.

You can use LabVIEW with little programming experience. LabVIEW uses terminology, icons, and ideas familiar to scientists and engineers and relies on graphical symbols rather than textual language to describe programming actions.

LabVIEW has extensive libraries of functions and subroutines for most programming tasks. For Windows, Macintosh, and Sun, LabVIEW contains application specific libraries for data acquisition and VXI instrument control. LabVIEW also contains application-specific libraries for GPIB and serial instrument control, data analysis, data presentation, and data storage. LabVIEW includes conventional program development tools, so you can set breakpoints, animate program execution to see how data passes through the program, and single-step through the program to make debugging and program development easier.

How Does LabVIEW Work?

LabVIEW includes libraries of functions and development tools designed specifically for instrument control. For Windows, Macintosh, and Sun, LabVIEW also contains libraries of functions and development tools for data acquisition. LabVIEW programs are called *virtual instruments (VIs)* because their appearance and operation imitate actual instruments. However, they are analogous to functions from conventional language programs. VIs have both an interactive user interface and a source code equivalent, and accept parameters from higher-level VIs. The following are descriptions of these three VI features.

- VIs contain an interactive user interface, which is called the *front panel*, because it simulates the panel of a physical instrument. The front panel can contain knobs, push buttons, graphs, and other controls and indicators. You input data using a keyboard and mouse, and then view the results on the computer screen.
- VIs receive instructions from a *block diagram*, which you construct in G. The block diagram supplies a pictorial solution to a programming problem. The block diagram contains the source code for the VI.
- VIs use a hierarchical and modular structure. You can use them as top-level programs, or as subprograms within other programs or subprograms. A VI within another VI is called a *subVI*. The *icon and connector pane* of a VI work like a graphical parameter list so that other VIs can pass data to it as a subVI.

With these features, LabVIEW promotes and adheres to the concept of *modular programming*. You divide an application into a series of tasks, which you can divide again until a complicated application becomes a series of simple subtasks. You build a VI to accomplish each subtask and then combine those VIs on another block diagram to accomplish the larger task. Finally, your top-level VI contains a collection of subVIs that represent application functions.

Because you can execute each subVI by itself, apart from the rest of the application, debugging is much easier. Furthermore, many low-level subVIs often perform tasks common to several applications, so that you can develop a specialized set of subVIs suited to applications you can construct.

Installing LabVIEW

For instructions on how to install LabVIEW, see your LabVIEW release notes.

After installing LabVIEW, the default configuration setting is correct for the purposes of this tutorial. If you would like to explore LabVIEW configuration options, see the *Preferences Dialog Boxes* section of Chapter 8, *Customizing Your LabVIEW Environment*, in the *LabVIEW User Manual*.

LabVIEW Files

The LabVIEW system consists of the LabVIEW application and a number of associated files.

LabVIEW uses several directories and files from the hard drive to store information necessary to create your VIs. These directories and files include, among others:

- The `vi.lib` directory. This directory contains libraries of VIs, such as analysis VIs.
- The `examples` directory. This directory contains many sample VIs that demonstrate LabVIEW's program functionality.
- The `tutorial.llb` library. This file, located in the `vi.lib` directory, contains a library of VIs that this tutorial uses.

You can access the contents of these files and directories from within the LabVIEW environment.

Virtual Instruments

LabVIEW programs are called virtual instruments (VIs). VIs have three main parts: the *front panel*, the *block diagram*, and the *icon/connector*.

OBJECTIVE To open, examine, and operate a VI, and to familiarize yourself with the basic concepts of a virtual instrument.

Front Panel

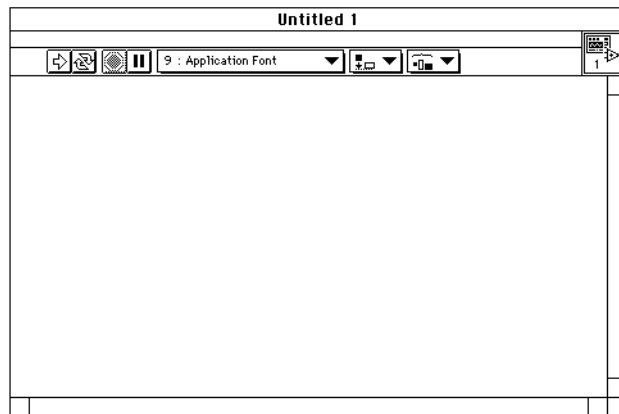
1. **(Windows)** Open LabVIEW by double-clicking with the mouse button on the LabVIEW icon in the LabVIEW group. If this is the first time you have opened LabVIEW, the program prompts you to enter your name, the name of your company, and your LabVIEW serial number.

(Macintosh) Launch LabVIEW by double-clicking on the LabVIEW icon in the LabVIEW folder. If this is the first time you have launched LabVIEW, the program prompts you to enter your name, the name of your company, and your LabVIEW serial number.

(UNIX) Launch LabVIEW by typing `labview <Return>` in a shell window. If LabVIEW is not in your executable path, you must type in the path to the LabVIEW executable followed by `labview`, as shown in the following example.

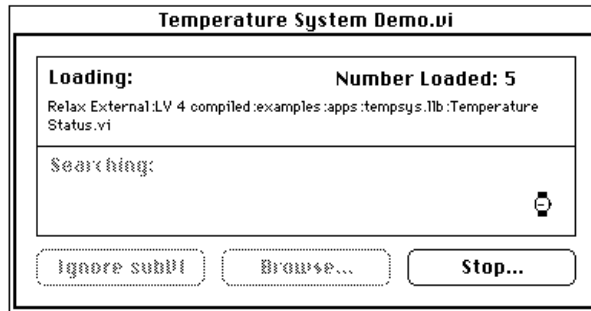
```
/usr/lib/labview/labview
```

(All Platforms) After a few moments, a blank, untitled front panel appears.



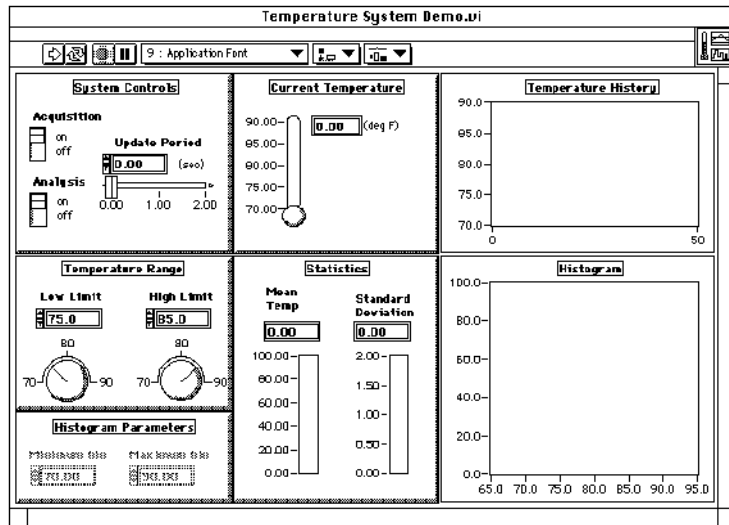
2. Open the Temperature System Demo VI by following these steps.
 - a. Select **File»Open**.
 - b. Double-click on examples. Double-click on apps. Double-click on tempsys.llb.
 - c. Double-click on Temperature System Demo.vi.

While the VI loads, a dialog box appears, which describes the name of the VI currently loading, the name of the hard drive that the VI is located on, the directories and paths being searched, and the number of the VI in the loading process. The following illustration shows the dialog box that appears when you load the Temperature System Demo VI.



After a few moments, the Temperature System Demo VI front panel appears, as the following illustration shows. The front panel contains

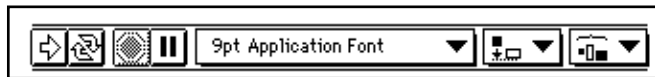
several numeric controls, Boolean switches, slide controls, knob controls, charts, graphs, and a thermometer indicator.



Front Panel Toolbar

The front panel contains a toolbar of command buttons and status indicators that you use for running and debugging VIs. It also contains font options and alignment and distribution options for editing VIs.

Front Panel Toolbar:



Run button—Runs the VI



Continuous run button—Runs the VI over and over; useful for debugging



Stop button—Aborts VI execution



Pause/Continue button—Pauses VI execution/Continues VI execution



Font ring—Sets font options, including font type, size, style, and color



Alignment ring—Sets alignment options, including vertical, top edge, left, and so on, for two or more objects



Distribution ring—Sets distribution options, including gaps, compression, and so on, for two or more objects



1. On the front panel, run the VI by clicking on the run button in the toolbar.



The button changes appearance to indicate that the VI is running.

The Temperature System Demo VI simulates a temperature monitoring application. The VI takes temperature readings and displays them in the thermometer indicator and on the chart. The Update Period slide controls how fast the VI acquires the new temperature readings. LabVIEW also plots high and low temperature limits on the chart, which you can change using the Temperature Range knobs in the middle left border. If the current temperature reading is out of the set range, LEDs light up next to the thermometer.

This VI continues to run until you click the Acquisition switch to Off. You can also turn the data analysis on and off. The analysis consists of a running calculation of the mean and standard deviation of the temperature values and a histogram of the temperature values.

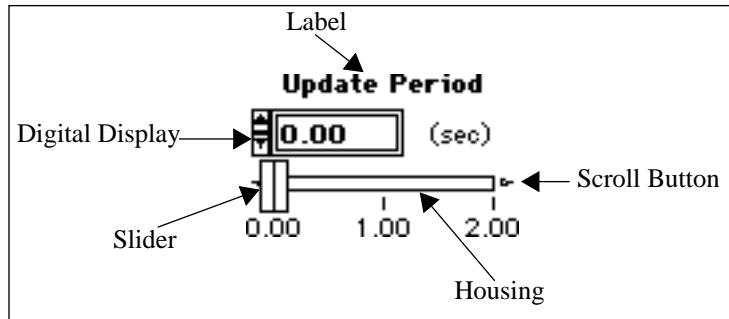


2. Use the Operating tool to change the values of the high and low limits. First, highlight the old value, either by double-clicking on the value you want to change, or by clicking and dragging across the value with the Labeling tool. When the initial value is highlighted, type a new value and press <Enter> (Windows); <return> (Macintosh); <Return> (Sun); or <Enter> (HP-UX). You can also click on the enter button in the toolbar, or click the mouse in an open area of the window to enter the new value.



3. Change the Update Period slide control, shown in the following illustration, by placing the Operating tool on the slider and dragging it to a new location.





4. Practice adjusting the other controls.
5. Stop the VI by clicking on the Acquisition switch. The VI may not stop immediately because the VI has to wait for the last equation or analysis set to complete operation.

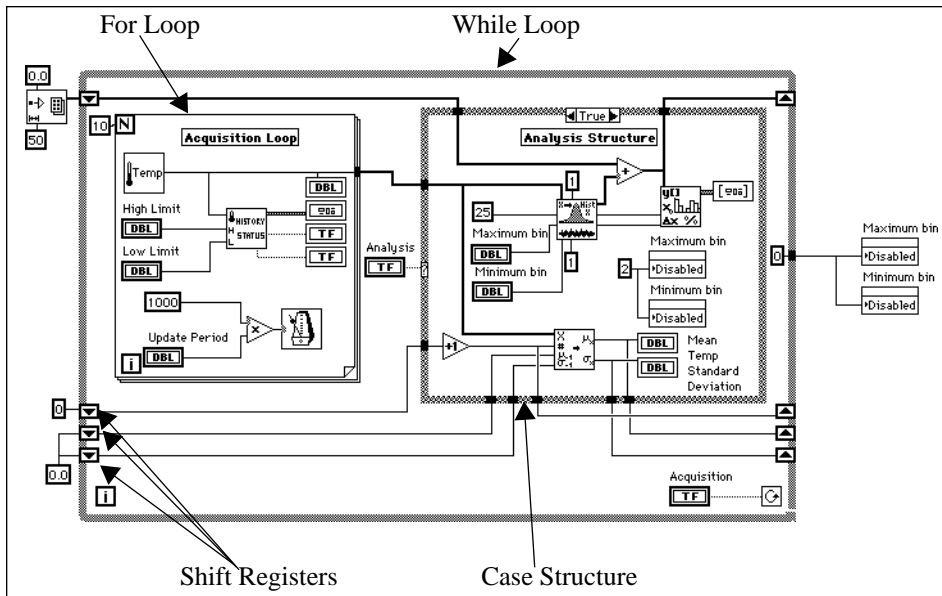
**Note:**

You should always wait for a VI to complete execution or you should design a method to stop it, such as placing a switch on the front panel. In this case, the VI collects the last set of temperature values, performs the analysis, and comes to a completion when you click the Acquisition switch.

Although the VI stops if you click on the stop button in the toolbar, this is not the best way to stop VIs because the stop button halts the program immediately. This may interrupt critical I/O functions, and so it is probably not desirable.

Block Diagram

The following block diagram represents a complete LabVIEW application, and is an example of how intricate LabVIEW programming can be. Subsequent chapters in this tutorial detail structures and elements mentioned in this section. It is not necessary to understand all of these block diagram elements at this time to appreciate the nature of a block diagram.



1. Open the block diagram of the Temperature System Demo VI by choosing **Windows»Show Diagram**.
2. Examine the different objects in the block diagram.

Each front panel has an accompanying block diagram, which is the VI equivalent of a program. You build the block diagram using the graphical programming language, G. You can think of the block diagram as source code. The components of the block diagram represent program nodes such as For Loops, Case structures, and multiplication functions. The components are *wired* together to show the flow of data within the block diagram.

The outermost structure is a While Loop. It continues to run what is inside of it until the Acquisition switch is set to Off. The arrow terminals on the border of the While Loop are called *Shift Registers* and store values from one iteration of the loop to the next. The values that the shift registers store here are the histogram, analysis iteration value, mean, and standard deviation, in that order.

The two main structures inside the While Loop are a For Loop and a Case structure. The acquisition of the data takes place inside the For Loop. The For Loop takes 10 temperature readings at the rate specified by Update Period and plots each reading on the thermometer and the chart. The VI also compares the temperature to the high and low limits.

The Case structure controls the temperature analysis. If the Analysis switch is off, the VI performs no analysis. You can see this by clicking on one of the arrows next to the word `True`. In the False case, no analysis takes place, and the histogram and analysis iteration value are reset to zero. Change back to the True case using the same method you used to change to the False case. Here, two subVIs analyze the data—one keeps a running mean and standard deviation of the temperatures, and the other keeps a running histogram of the acquired temperatures.

You do not need to fully understand all of the structures at this point. The subsequent chapters in this tutorial discuss in greater detail each element that appears in this VI.

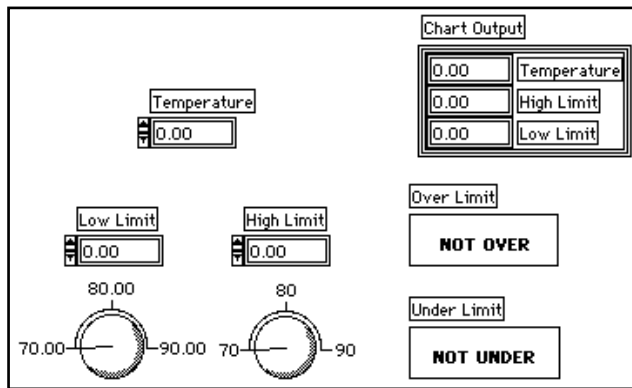
Hierarchy

The power of LabVIEW lies in the hierarchical nature of VIs. After you create a VI, you can use it as a *subVI* in the block diagram of a higher level VI. You can have an essentially unlimited number of layers in the hierarchy.

As an example, look at a VI that the Temperature System Demo VI uses as a subVI in its block diagram.



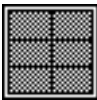
1. Open the Temperature Status subVI by double-clicking on the subVI icon. The following front panel appears.



Icon/Connector Pane



Icon



Connector



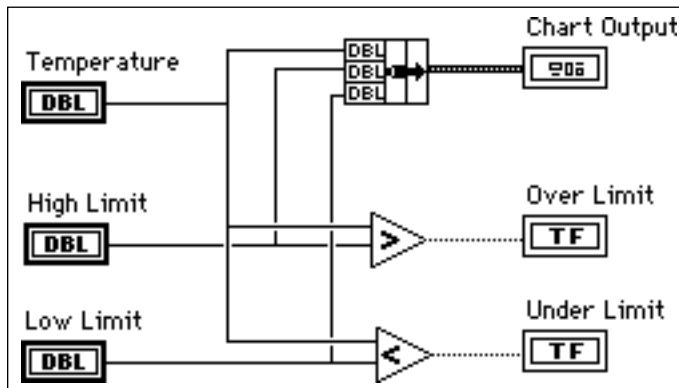
You use the icon/connector pane to turn a VI into an object that you can use in the block diagrams of other VIs as a subroutine or function. The icon and connector are located in the upper right corner of the VI front panel. The icon graphically represents the VI in the block diagram of other VIs. The connector *terminals* determine where you must wire the inputs and outputs on the icon. The terminals are analogous to parameters of a subroutine or function. They correspond to the controls and indicators on the front panel of the VI. The icon hides the connector until you choose to view it.

2. Put the Operating tool on the icon pane in the upper right corner of the front panel and pop up. A pop-up menu appears.
3. Select **Show Connector** from the pop-up menu. The cursor changes to the Wiring tool, shown on the left.

The squares on the connector are terminals that correspond to the controls and indicators on the front panel.

4. Click on a terminal. It turns black. Notice that a control or indicator becomes highlighted on the front panel. When you wire the control or indicator (terminal), the data in it passes to (or is received from) the other end of the wire.

5. Place the Wiring tool on the connector in the front panel and pop up. A pop-up menu appears.
6. Select **Show Icon**. The Wiring tool changes back to the Operating tool.
7. Switch to the block diagram by selecting **Windows»Show Diagram**. At this time, you do not need to understand what all the parts of the block diagram do. It is enough to notice that a subVI can be complex or simple in itself.



By creating subVIs, you can make your block diagrams modular. This modularity makes VIs easy to debug, understand, and maintain.

8. Switch to the front panel (**Windows»Show Panel**).
9. Select **File»Close** and do not save any changes you have made.

Tools Palette

LabVIEW uses a floating **Tools** palette, which you can use to edit and debug VIs. You use the <Tab> key to tab through the commonly used tools on the palette. If you have closed the **Tools** palette, select **Windows»Show Tools Palette** to display the palette. The following illustration displays the **Tools** palette.



Operating tool—Places **Controls** and **Functions** palette items on the front panel and block diagram



Positioning tool—Positions, resizes, and selects objects



Labeling tool—Edits text and creates free labels



Wiring tool—Wires objects together in the block diagram



Object pop-up menu tool—Brings up a pop-up menu for an object



Scroll tool—Scrolls through the window without using the scrollbars



Breakpoint tool—Sets breakpoints on VIs, functions, loops, sequences, and cases



Probe tool—Creates probes on wires



Color copy tool—Copies colors for pasting with the Color tool



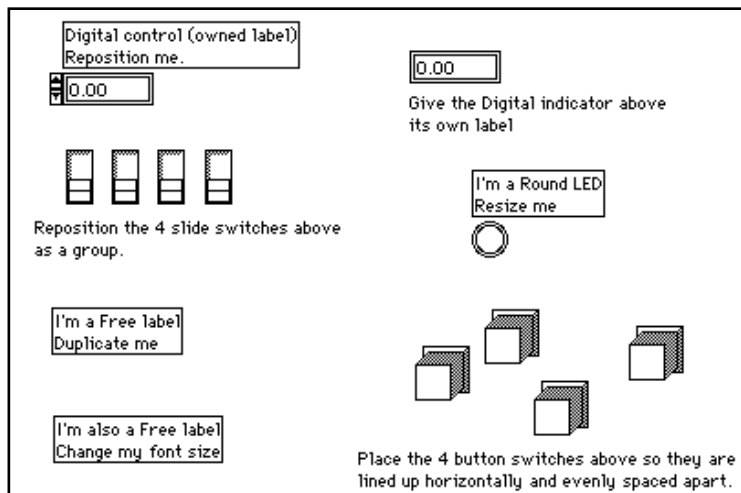
Color tool—Sets foreground and background colors

Editing Techniques

OBJECTIVE To learn LabVIEW editing techniques.

To work on the Editing Exercise VI, select **File»Open**. The Editing Exercise VI is located in `examples\general\controls\smplctls.llb`.

The front panel of the Editing Exercise VI contains a number of LabVIEW objects. Your objective is to change the front panel of the VI as the following illustration shows.



1. If the **Tools** palette is not visible, select **Windows»Show Tools Palette** to display it.
2. Reposition the digital control.
 - a. Choose the Positioning tool from the **Tools** palette.
 - b. Click on the digital control and drag it to another location.

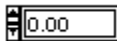


Notice that the label follows the control. The control *owns* the label.

3. Click on a blank space on the front panel to deselect the control, and then click on the label and drag it to another location.

Notice that the control does not follow the label. You can position an owned label anywhere relative to the control, but when the control moves, the label follows.

4. Switch to the block diagram by selecting **Windows»Show Diagram**.
5. Move the block diagram so that you can see both windows.
6. Click on the front panel to make it active.
7. Copy the digital control to the block diagram to create a corresponding constant.



- a. Choose the Positioning tool from the **Tools** palette.
- b. Click on the digital control. While holding the mouse button down, drag the digital control to the block diagram. The digital control now appears as a corresponding constant on the block diagram. You can also use the **Copy** and **Paste** options from the **Edit** menu to copy the control and then paste it to the block diagram.

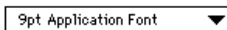


Note: *You can use this same process to drag or copy a constant from the block diagram to the front panel to create a corresponding control.*

8. Reposition the four slide switches as a group.
 - a. Using the Positioning tool, click in an open area near the four switches, hold down the mouse button, and drag until all the switches lie within the selection rectangle.
 - b. Click on the selected switches and drag them to a different location.
9. Duplicate the free label. Using the Positioning tool, hold down <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX), click on the free label, and drag the duplicate of the free label to a new location. For UNIX, you can use the middle mouse button to drag the label. This creates a duplicate copy of the label.



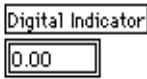
10. Change the font size of the free label.



- a. Select the text by using the Labeling tool. You can triple-click on the text, or click on the first letter in the label and drag the cursor to the end of the label.
- b. Change the selected text size to 12 points by choosing **Size** from the Font ring, located in the toolbar.

11. Create an owned label for the digital indicator.

- a. Pop up on the digital indicator and choose **Show»Label** from the pop-up menu.
- b. Type Digital Indicator inside the bordered box and click the mouse button outside the label. You can also end text entry by pressing <Enter> on the numeric keypad.



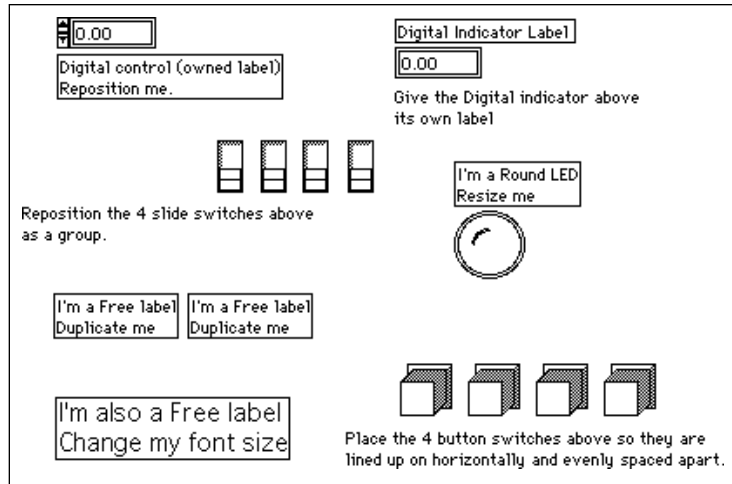
12. Resize the round LED. Place the Positioning tool over a corner of the LED until the tool becomes the Resizing cursor. Click and drag the cursor outward to enlarge the LED. If you want to maintain the current ratio of horizontal to vertical size of the LED, hold down the <Shift> key while resizing the LED.



13. Change the color of the round LED.
 - a. Using the Color tool, pop up on the LED.
 - b. Choose a color from the selection palette. When you release the mouse button, the object assumes the last color you selected.
14. Place the four button switches so they are aligned horizontally and evenly spaced.



- a. Using the Positioning tool, click in an open area near the four switches and drag until all the switches lie within the selection rectangle.
- b. Align the switches horizontally by clicking on the Alignment ring in the toolbar and choosing the Vertical Centers alignment.
- c. Space the switches evenly by clicking on the Distribution ring and choosing the Horizontal Centers distribution. The front panel should now look similar to the following illustration.



15. Close the VI by selecting **File»Close**. Do not save any changes.

Controls Palette

The **Controls** palette consists of a graphical, floating palette that automatically opens when you launch LabVIEW. You use this palette to place controls and indicators on the front panel of a VI. Each top-level icon contains subpalettes. If the **Controls** palette is not visible, you can open the palette by selecting **Windows»Show Controls Palette** from the front panel menu. You can also pop up on an open area in the front panel to access a temporary copy of the **Controls** palette.

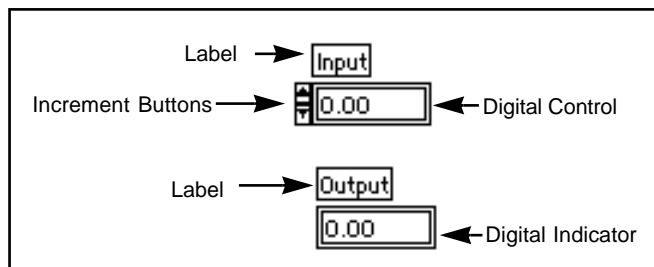
The following illustration displays the top-level of the **Controls** palette.



Controls and Indicators

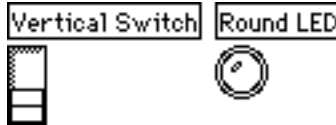
Numeric Controls and Indicators

You use numeric controls to enter numeric quantities, while numeric indicators display numeric quantities. The two most commonly used numeric objects are the *digital control* and the *digital indicator*.



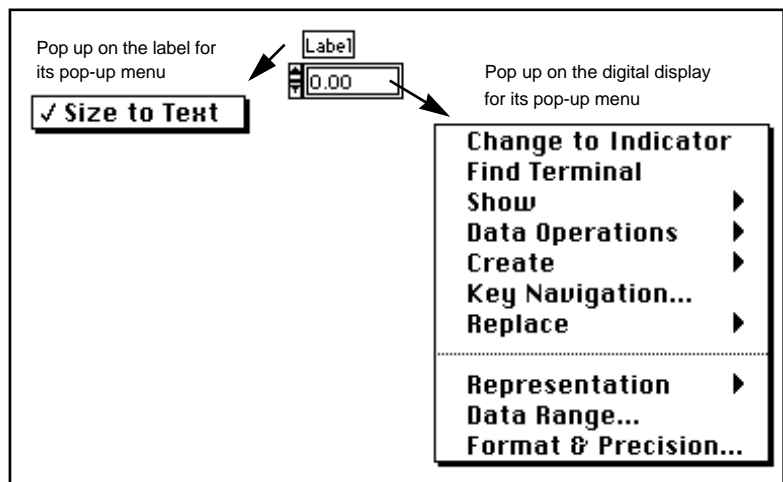
Boolean Controls and Indicators

You use Boolean controls and indicators for entering and displaying Boolean (True/False) values. Boolean objects simulate switches, buttons, and LEDs. The most commonly used Boolean objects are the *vertical switch* and the *round LED*.



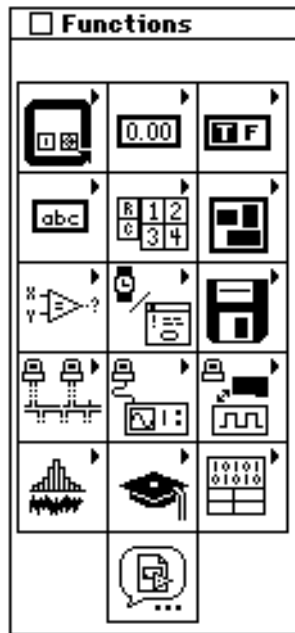
Configuring Controls and Indicators

You can configure nearly all the controls and indicators using options from their pop-up menus. *Popping up on individual components of controls and indicators displays menus for customizing those components.* An easy way to access the pop-up menu is to click the Object pop-up menu tool, shown at left, on any object that has a pop-up menu. The following picture illustrates this display method for a digital control.



Functions Palette

The **Functions** palette consists of a graphical, floating palette that automatically opens when you switch to the block diagram. You use this palette to place nodes (constants, indicators, VIs, and so on) on the block diagram of a VI. Each top-level icon contains subpalettes. If the **Functions** palette is not visible, you can select **Windows»Show Functions Palette** from the block diagram menu to display it. You can also pop up on an open area in the block diagram to access a temporary copy of the **Functions** palette. The following illustration displays the top-level of the **Functions** palette.



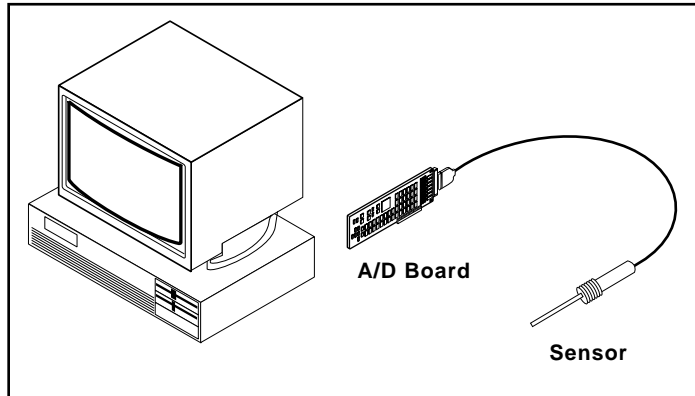
Building a VI

OBJECTIVE To build a VI that simulates acquisition of a temperature reading.

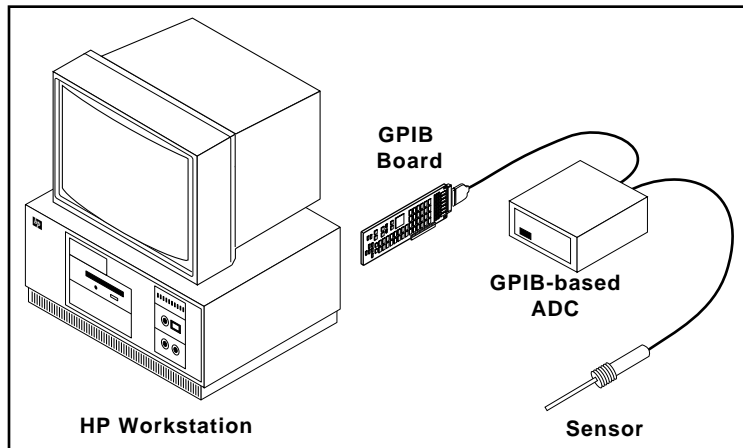
You will use the Demo Voltage Read VI to measure the voltage, and then multiply the reading by 100.0 to convert the voltage into a temperature (in degrees F).

Imagine that you have a transducer or sensor that converts temperature to voltage.

(Windows, Macintosh, and Sun) The sensor connects to an analog-to-digital converter (A/D) board, as shown in the following illustration, which converts voltage to digital data.



(HP-UX) The sensor could also be connected to an analog-to-digital converter that is connected to the computer through GPIB, as shown in the following illustration. This also converts voltage to digital data.



Front Panel

1. Open a new front panel by selecting **File»New**. For Windows and UNIX, if you have closed all VIs, select **New VI** from the LabVIEW dialog box.



Note: *If the Controls palette is not visible, select **Windows»Show Controls Palette** to display the palette. You can also access the Controls palette by popping up in an open area of the front panel.*



2. Select a Thermometer indicator from **Controls»Numeric**, and place it on the front panel.
3. Type Temp inside the label text box and click on the enter button on the toolbar.

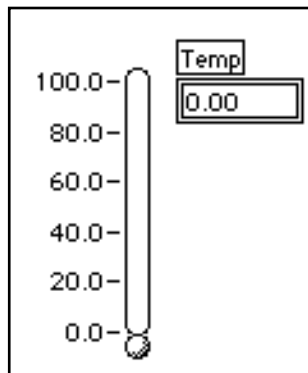


Note: *If you click outside the text box without entering text, the label disappears. You can show the label again by popping up on the control and selecting **Show»Label**.*

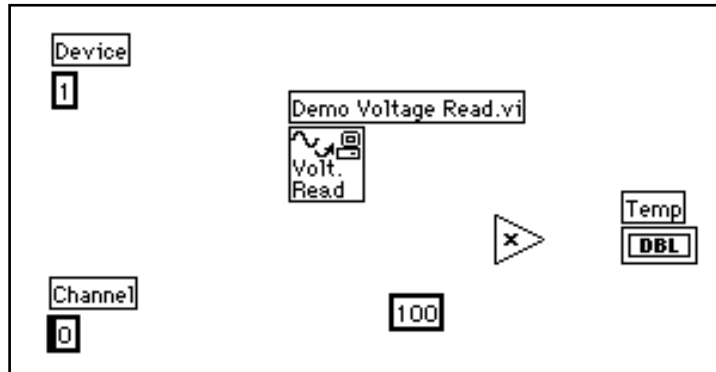
Remember, to pop up, use right-click (<command>-click on Macintosh).



4. Rescale the thermometer control to display the temperature between 0.0 and 100.0.
 - a. Using the Labeling tool, double-click on 10.0 in thermometer scale to highlight it.
 - b. Type 100.0 in the scale and click the mouse button anywhere outside the display window. LabVIEW automatically scales the intermediary increments. The temperature control should now look like the following illustration.



Block Diagram



1. Open the block diagram by choosing **Windows»Show Diagram**. Select the block diagram objects from the **Functions** palette. For each object that you want to insert, select the icon and then the object from the top-level of the palette, or choose the object from the appropriate subpalette. When you position the mouse on the block diagram, LabVIEW displays an outline of the object. This is your opportunity to place the object where you want on the block diagram.



Note:

*If the Functions palette is not visible, select **Windows»Show Functions Palette** to display the palette. You can also access the **Functions palette** by popping up in an open area of the block diagram.*



The Demo Voltage Read VI (**Functions»Tutorial**) simulates reading a voltage from a plug-in data acquisition board.



Multiply function (**Functions»Numeric**). In this exercise, the function multiplies the voltage returned by the Demo Voltage Read VI by 100.0.

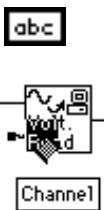


Numeric Constant (**Functions»Numeric**). You need two numeric constants: one for the scaling factor of 100 and one for the device constant. For the first numeric constant, type 100.0 when the constant first appears on the block diagram.

2. Create the second numeric constant using a shortcut to automatically create and wire the constant to the Demo Voltage Read VI.
 - a. Using the Wiring tool, pop up on the input marked Board ID on the Demo Voltage Read VI and select **Create Constant** from the pop-up menu. This option automatically creates a numeric constant and wires it to the Demo Voltage Read VI.
 - b. Type 1 when the constant first appears on the block diagram. This changes the default value of zero to one. Notice that you do not have to change to the Labeling tool to insert the value when using this feature.
 - c. Using the Labeling tool, change the default label (Board ID) to Device.



In this example, the two numerics represent the constant 100.0 and the device for the multiply function.



String Constant (**Functions»String**).

3. Pop up on the input marked Channel, at the bottom left of the Demo Voltage Read VI and select **Create Constant** from the pop-up menu. This option automatically creates a string constant and wires it to the Demo Voltage Read VI.
4. Type 0 when the constant first appears on the block diagram. Notice that in this instance, Channel appears in the default label so you do not have to change the label.

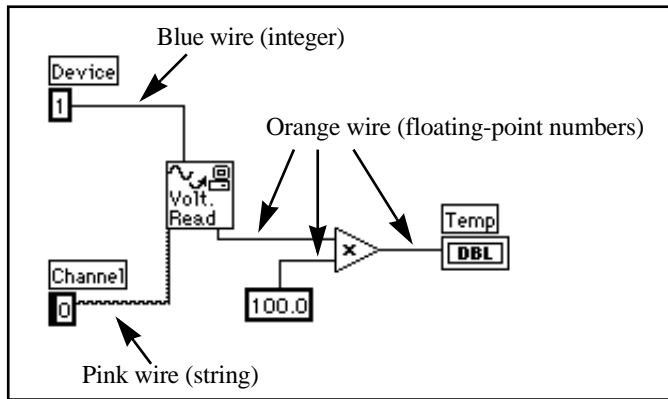
In this example, you use the string constant to represent the channel number.



Note: *You can create and wire controls, constants and indicators with most functions. If these options are not available for a particular function, the **Create Control**, **Create Constant** and **Create Indicator** options are disabled on the pop-up menu. For more information on this feature, see the *Create & Wire Controls, Constants, and Indicators* section later in this chapter.*

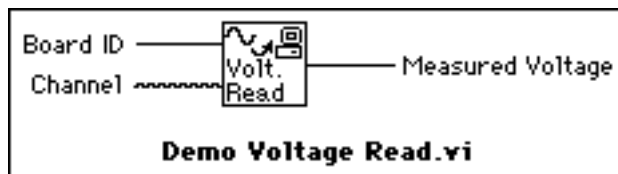


- Using the Wiring tool, wire the remaining objects together as explained in the *Wiring Techniques* section later in this chapter.

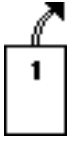


LabVIEW color keys wires to the kind of data each wire carries. Blue wires carry integers, orange wires carry floating-point numbers, green wires carry Booleans, and pink wires carry strings.

You can activate the Help window by choosing **Help»Show Help**. Placing any of the editing tools on a node displays the inputs and outputs of that function in the Help window. As you pass an editing tool over the VI icon, LabVIEW highlights the wiring terminals in both the block diagram and the Help window. When you begin to wire your own diagrams, this flashing highlight can help you to connect your inputs and outputs to the proper terminals.



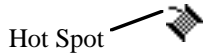
The Demo Voltage Read VI simulates reading the voltage at Channel 0 of a plug-in board. The VI then multiplies the voltage by 100.0 to convert it to a temperature in °F.



Wiring Techniques

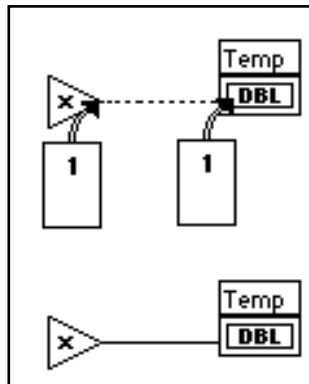
In the wiring illustrations in this section, the arrow at the end of this mouse symbol shows where to click and the number printed on the mouse button indicates how many times to click the mouse button.

The *hot spot* of the tool is the tip of the unwound wiring segment.



To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and click on the second terminal. It does not matter at which terminal you start.

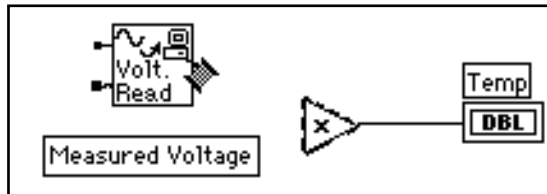
When the Wiring tool is over a terminal, the terminal area blinks, to indicate that clicking connects the wire to that terminal. *Do not* hold down the mouse button while moving the Wiring tool from one terminal to another. You can bend a wire once by moving the mouse perpendicular to the current direction. To create more bends in the wire, click the mouse button. To change the direction of the wire, press the spacebar. Click with the mouse button, to *tack* the wire down and move the mouse perpendicularly.



Tip Strips



When you move the Wiring tool over the terminal of a node, a tip strip for that terminal pops up. Tip strips consist of small, yellow text banners that display the name of each terminal. These tip strips should help you to wire the terminals. The following illustration displays the tip strip (Measured Voltage) that appears when you place the Wiring tool over the output of the Demo Voltage Read VI.



Note:

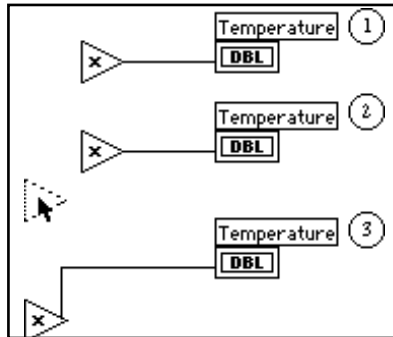
When you place the Wiring tool over a node, LabVIEW displays wire stubs that indicate each input and output. The wire stub has a dot at its end if it is an input to the node.

Showing Terminals

It is important that you wire the correct terminals of a function. You can show the icon connector to make correct wiring easier. To do this, pop up on the function and choose **Show»Terminals**. To return to the icon, pop up on the function and again select **Show»Terminals**.

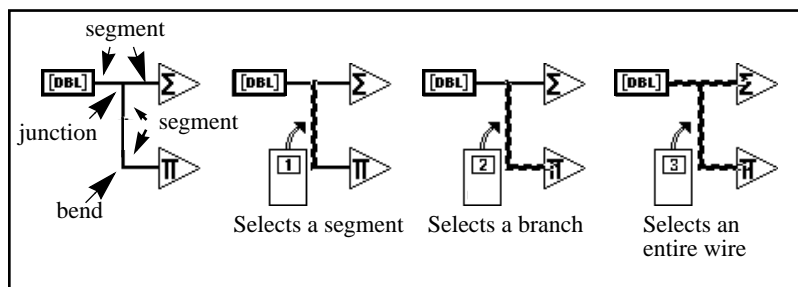
Wire Stretching

You can move wired objects individually or in groups by dragging the selected objects to a new location with the Positioning tool.



Selecting and Deleting Wires

You may accidentally wire nodes incorrectly. If you do, select the wire you want to delete and then press <Delete>. A wire *segment* is a single, horizontal or vertical piece of wire. The point where three or four wire segments join is called a *junction*. A wire *branch* contains all the wire segments from one junction to another, from a terminal to the next junction, or from one terminal to another if there are no junctions in between. You select a wire segment by clicking on it with the Positioning tool. Double-clicking selects a branch, and triple-clicking selects the entire wire.



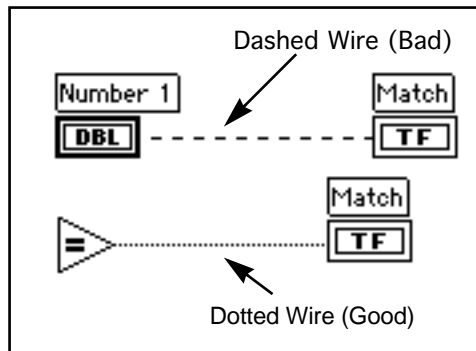
Bad Wires



A dashed wire represents a bad wire. You can get a bad wire for a number of reasons, such as connecting two controls, or connecting a source terminal to a destination terminal when the data types do not match (for instance, connecting a numeric to a Boolean). You can remove a bad wire by clicking on it with the Positioning tool and pressing <Delete>. Choosing **Edit»Remove Bad Wires** deletes all bad wires in the block diagram. This is a useful quick fix to try if your VI refuses to run or returns the Signal has loose ends error message.



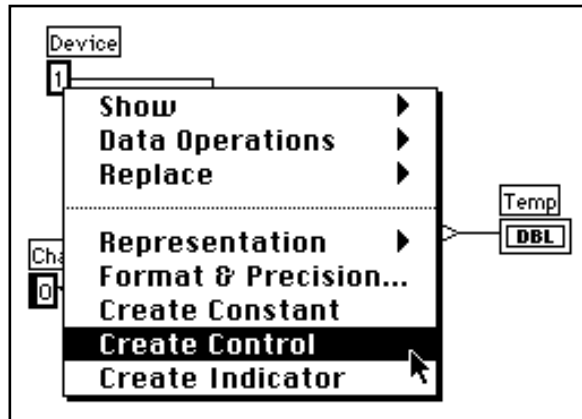
Note: *Do not confuse a dashed wire with a dotted wire. A dotted wire represents a Boolean data type, as the following illustration shows.*



Create & Wire Controls, Constants, and Indicators

For terminals acting as inputs on the block diagram, LabVIEW has two features that you can use to create and wire a control or constant. You access these features by popping up on the terminal and choosing **Create Control** or **Create Constant**. LabVIEW automatically creates

and wires the correct control or constant type to the terminal input. The following illustration shows an example pop-up menu.



For a terminal acting as an output on the block diagram, you can choose the **Create Indicator** feature to create and then wire an indicator to the terminal. You access this feature by popping up on the terminal and choosing **Create Indicator**. LabVIEW automatically creates and wires the correct indicator type to the output of a terminal.

Run the VI

1. For Windows and Macintosh, make the front panel active by clicking anywhere on it. In UNIX, make the front panel active by clicking on the window title bar or by choosing **Windows»Show Panel**.
2. Run the VI by clicking on the run button in the toolbar of the front panel.



Notice that you have to rerun the VI each time. If you want to repeatedly run the VI, you must click on the continuous run button.



3. Click on the continuous run button in the toolbar.
4. Click on the continuous run button again to deselect it. The VI then completes execution and quits.



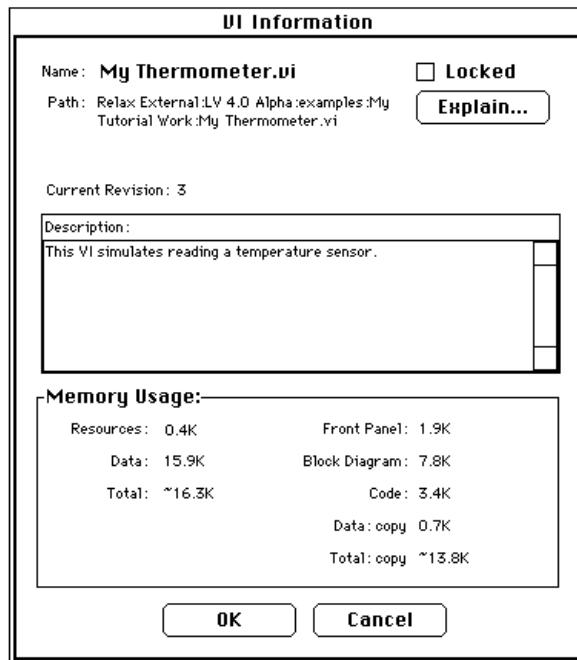
Note:

The continuous run button is not the preferred method for repeating block diagram code. You should use a looping structure. This is covered in Chapter 3, Loops and Charts, of this tutorial.

Documenting the VI

You can document the VI by choosing **Windows»Show VI Info...** Type the description of the VI in the VI Information dialog box. You can then recall the description by again selecting **Windows»Show VI Info...**

1. Document the VI. Select **Windows»Show VI Info...** Type the description for the VI, as shown in the following illustration, and click on **OK**.



You can view the descriptions of objects on the front panel (or their respective terminals on the block diagram) by popping up on the object and choosing **Description...**. The location of this choice differs between the front panel and block diagram.

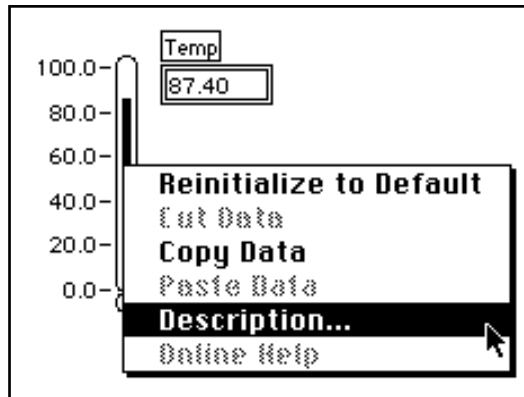
front panel: Pop up on the object and choose **Data Operations»Description...**

block diagram: Pop up on the object and choose **Description...**

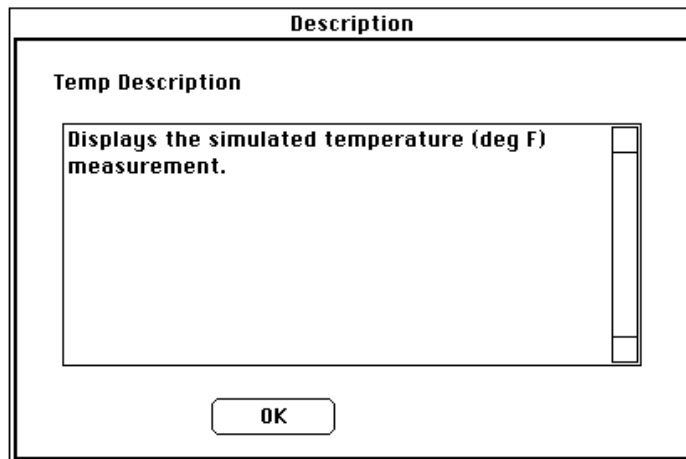


Note: *You cannot change the description while running a VI.*

The following illustration is an example pop-up menu that appears while you are running a VI. You cannot add to or change the description while running the VI, but you can view any previously entered information.



2. Document the thermometer indicator.
 - a. On the front panel, pop up on the thermometer indicator and choose **Data Operations»Description...**
 - b. Type the description for the indicator, as shown in the following illustration, and click on **OK**.



3. Show the description you created again by popping up on the thermometer indicator and selecting **Data Operations» Description...**

Saving and Loading VIs

As with other applications, you can save your VI to a file in a regular directory. With LabVIEW, you can also save multiple VIs in a single file called a *VI library*. The `tutorial1.llb` library is an example of a VI library.

If you are using Windows 3.1, you should save your VIs into VI libraries because you can use long file names (up to 255 characters) with mixed cases.

Otherwise, you should not use VI libraries unless you need to transfer your VIs to Windows 3.1. Saving VIs as individual files is more effective because you can copy, rename, and delete files more easily than if you are using a VI library. For a list of the advantages and disadvantages of using VI libraries and individual files, see the *Saving VIs* section in Chapter 2, *Creating VIs*, of the *LabVIEW User Manual*.

Even though you may not save your own VIs in VI libraries, you should be familiar with how they work. For that reason, you should save all VIs that you create during this tutorial into VI libraries to become familiar with using these libraries.

Save your VI in a VI library.

1. Select **File»Save As...** If you are using UNIX, specify a location in the file system where you have write privileges. For example, you might select your home directory.
2. Do not save your files in the `examples` directory. Instead, create your own directory and label it `Tutorial VIs`.
3. Create the VI library.

(Windows) Select **New...** or the **New VI Library** button to create the VI Library.

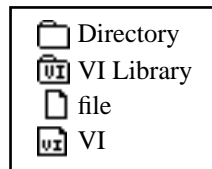
(Macintosh) If you use the native file dialog box, **Edit» Preferences...**, select **Use LLBs** to access LabVIEW's file dialog box. To create the VI library, click on **Save»New...**

(UNIX) Select **Save»New...**

4. Enter `mywork` as the name of the new library in the dialog box and click on the **VI Library** button. The library name must be followed by an `.llb` extension. For Windows 3.1, you must limit your library names to eight characters or less. LabVIEW appends the `.llb` extension if you do not include it.

VI libraries have the same load, save, and open capabilities as directories. VI libraries, however, are not hierarchical. That is, you cannot create a VI library inside of another VI library. You cannot create a new directory inside a VI library, either. There is no way to list the VIs in a VI library outside of the LabVIEW environment.

After you create a VI library, it appears in LabVIEW's file dialog box as a folder with VI on the folder icon. Regular directories appear as a folder without the VI label.



5. Name the VI and save it in your new library. Look at the name in the ring control at the top of the dialog box. Make sure it is `mywork.llb`. If it is not, click on `mywork.llb` in the directory list to make sure you save your VI in the right place.
 - a. Type `My Thermometer.vi` in the dialog box.
 - b. Click on **OK**.
6. Close the VI by selecting **File»Close**.

Summary

Virtual instruments (VIs) have three main parts: the front panel, the block diagram, and the icon/connector. The front panel specifies the inputs and outputs of the VI. The block diagram consists of the executable code that you create using nodes, terminals, and wires. With the icon/connector, you can use a VI as a subVI in the block diagram of another VI.

The **Tools** palette consists of a graphical, floating palette. On the front panel and block diagram, you use tools from the **Tools** palette to build, edit, and debug VIs. You use the <Tab> key to tab through the commonly used tools on the palette. The most commonly used tools are:



Operating tool



Positioning tool



Labeling tool



Wiring tool



Color tool

You use the Operating tool to manipulate front panel controls and indicators. You use the Positioning tool to position, resize, and select objects. You use the Labeling tool to create free labels and to enter text in labels. You use the Wiring tool to wire objects together in the block diagram. You use the Color tool to set the foreground and background color of windows, controls, indicators, and so on.

The front panel and block diagram contain toolbars, which display the run button along with other buttons that control the execution of the VI.

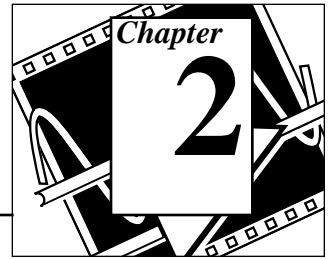
In the front panel, you place controls and indicators to denote the inputs and outputs of the VI. You use the **Controls** palette to add controls and indicators to the front panel. The **Controls** palette automatically pops up on the front panel when you launch LabVIEW. You can also access the **Controls** palette by selecting **Windows»Show Controls Palette**. Controls and indicators have different options that are configured from their pop-up menus. See the *LabVIEW User Manual* for more information regarding front panel controls and indicators.

The block diagram contains buttons, which also include features that you can use for execution debugging and single-stepping through VIs.

On the block diagram, you develop your source diagram by connecting nodes and terminals using the Wiring tool. You use the **Functions** palette to place nodes (structures, functions, and subVIs) on the block diagram. The **Functions** palette automatically pops up on the block diagram when you open the block diagram. You can also open the **Functions** palette by selecting **Windows»Show Functions Palette**. LabVIEW automatically places terminals, which are associated with the front panel controls and indicators on the block diagram. See the *LabVIEW User Manual* for more information concerning block diagram programming.

You can modify nearly all LabVIEW objects through their own pop-up menus. You access the pop-up menu by popping up on the object or by using the Object pop-up menu tool.

Popping up on individual components of an object accesses their own pop-up menus. So remember—when in doubt, pop up!



Creating a SubVI

You Will Learn:

- What a subVI is.
- How to create the icon and connector.
- How to use a VI as a subVI.

Understanding Hierarchy

One of the keys to creating LabVIEW applications is understanding and using the hierarchical nature of the VI. After you create a VI, you can use it as a *subVI* in the block diagram of a higher-level VI. Therefore, a subVI is analogous to a subroutine in C. Just as there is no limit to the number of subroutines you can use in a C program, there is no limit to the number of subVIs you can use in a LabVIEW program. You can also call a subVI inside another subVI.

When creating an application, you start at the top-level VI and define the inputs and outputs for the application. Then, you construct subVIs to perform the necessary operations on the data as it flows through the block diagram. If a block diagram has a large number of icons, group them into a lower-level VI to maintain the simplicity of the block diagram. This modular approach makes applications easy to debug, understand, and maintain.

Creating the SubVI

OBJECTIVE To make an icon and connector for the My Thermometer VI you created in Chapter 1 and use the VI as a subVI.

To use a VI as a subVI, you must create an icon to represent it on the block diagram of another VI, and a connector pane to which you can connect inputs and outputs.

Icon



Create the icon, which represents the VI in the block diagram of other VIs. An icon can be a pictorial representation of the purpose of the VI, or it can be a textual description of the VI or its terminals.



1. If you have closed the My Thermometer VI, open it by selecting **File»Open...**
2. Select `My Thermometer.vi` from `mywork.llb`.
3. Invoke the Icon Editor by popping up in the icon pane in the upper right corner of the front panel and choosing **Edit Icon**. As a shortcut, you can also double-click on the icon pane to edit the icon.

Icon Editor Tools and Buttons

The tools to the left of the editing area perform the following functions:



Pencil tool

Draws and erases pixel by pixel.



Line tool

Draws straight lines. Press <Shift> and then drag this tool to draw horizontal, vertical, and diagonal lines.



Dropper tool

Copies the foreground color from an element in the icon.



Fill bucket tool

Fills an outlined area with the foreground color.



Rectangle tool

Draws a rectangular border in the foreground color. Double-click on this tool to frame the icon in the foreground color.



Filled rectangle tool

Draws a rectangle bordered with the foreground color and filled with the background color. Double-click to frame the icon in the foreground color and fill it with the background color.



Select tool

Selects an area of the icon for moving, cloning, or other changes.



Text tool

Enters text into the icon design.



Foreground/ Background Displays the current foreground and background colors. Click on each to get a color palette from which you can choose new colors.

The buttons at the right of the editing screen perform the following functions:

Undo Cancels the last operation you performed.

OK Saves your drawing as the VI icon and returns to the front panel.

Cancel Returns to the front panel without saving any changes.



4. Erase the default icon.

a. With the Select tool, select the interior section of the default icon, shown at left.

a. Press <Delete> to erase the interior of the default icon.

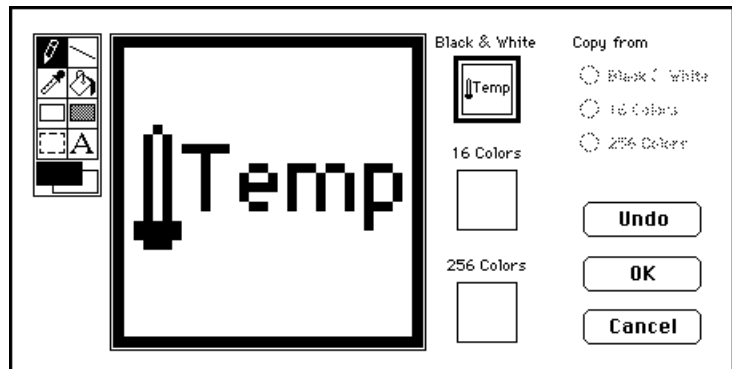


5. Draw the thermometer with the Pencil tool.

6. Create the text with the Text tool. To change the text font, double-click on the Text tool. Experiment with the editor.



Your icon should look similar to the following illustration.

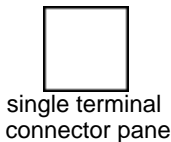
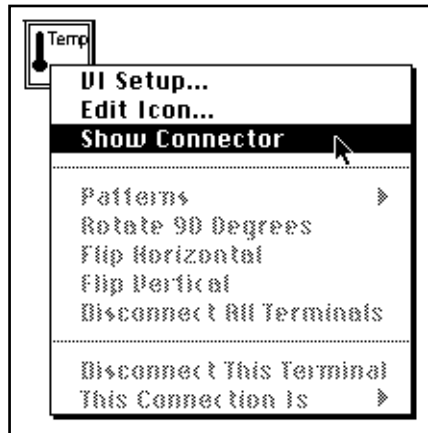


7. Close the Icon Editor by clicking on **OK** once you complete your icon. The new icon appears in the icon pane in the upper right corner of the front panel.

Connector

Now, you can create the connector.

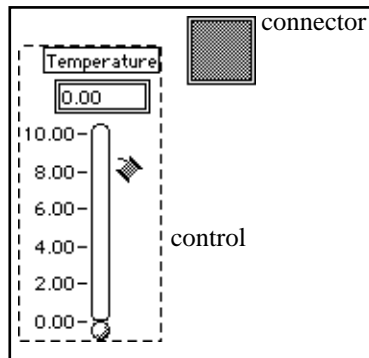
1. Define the connector terminal pattern by popping up in the icon pane on the front panel and choosing **Show Connector**, as the following illustration shows.



Because LabVIEW selects a terminal pattern based on the number of controls and indicators on the front panel, there is only one terminal—the thermometer indicator.

2. Assign the terminal to the thermometer.
 - a. Click on the terminal in the connector. The cursor automatically changes to the Wiring tool, and the terminal turns black.

- b. Click on the thermometer indicator. A moving dashed line frames the indicator, as the following illustration shows.



If you click in an open area on the front panel, the dashed line disappears and the selected terminal dims, indicating that you have assigned the indicator to that terminal. If the terminal is white, you have not made the connection correctly. Repeat the previous steps if necessary.

3. Save the VI by choosing **File»Save**. On the Macintosh, if you are using the native file dialog box to save into a VI library, you must click on the **Use LLBs** button before selecting the VI library.

This VI is now complete and ready for use as a subVI in other VIs. The icon represents the VI in the block diagram of the calling VI. The connector (with one terminal) outputs the temperature.



Note:

The connector specifies the inputs and outputs to a VI when you use it as a subVI. Remember that front panel controls can be used as inputs only; front panel indicators can be used as outputs only.

4. Close the VI by choosing **File»Close**.

Using a VI as a SubVI

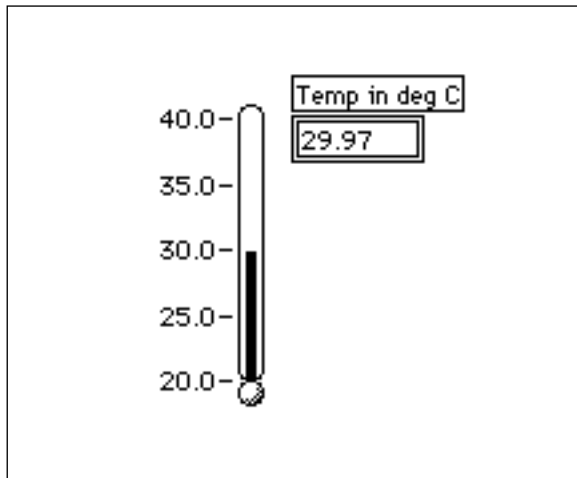
You can use any VI that has an icon and a connector as a subVI in the block diagram of another VI. You select VIs to use as subVIs from **Functions»Select a VI...** Choosing this option produces a file dialog box, from which you can select any VI in the system. If you open a VI that does not have an icon and a connector, a blank, square box appears in the calling VI's block diagram. You cannot wire to this node.

A subVI is analogous to a subroutine. A subVI node (icon/connector) is analogous to a subroutine call. The subVI 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.

OBJECTIVE To build a VI that uses the My Thermometer VI as a subVI.

The My Thermometer VI you built returns a temperature in degrees Fahrenheit. You will take that reading and convert the temperature to degrees Centigrade.

Front Panel



1. Open a new front panel by selecting **File»New**.

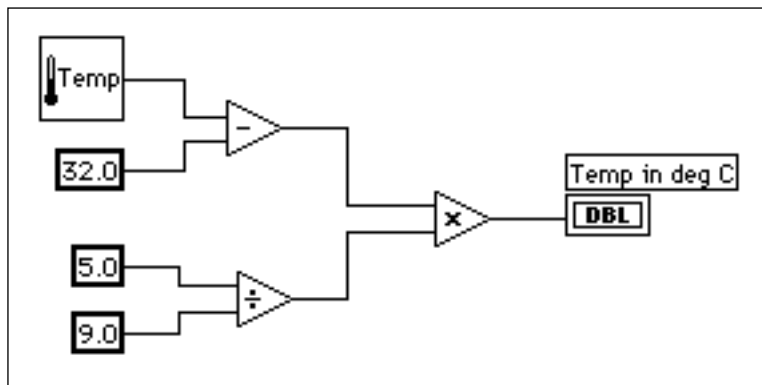


2. Choose the thermometer from **Controls»Numeric**. Label it Temp in deg C.
3. Change the range of the thermometer to accommodate the temperature values. With the Operating tool, double-click on the lower limit, type 20, and press <Enter> on the numeric keypad. You do not have to type the decimal and trailing zeroes. LabVIEW adds them automatically when you enter the value. Similarly, change the upper limit of the thermometer to 40 and press <Enter> on the numeric keypad. LabVIEW automatically adjusts the intermediate values.

Each time you create a new control or indicator, LabVIEW creates the corresponding terminal in the block diagram. The terminal symbols suggest the data type of the control or indicator. For example, a DBL terminal represents a double-precision, floating-point number.

Block Diagram

1. Select **Windows»Show Diagram**.
2. Pop up in a free area of the block diagram and choose **Functions»Select a VI...**A dialog box appears. Locate and open the mywork.llb library. Double-click on My Thermometer.vi or highlight it and click on **Open** in the dialog box. LabVIEW places the My Thermometer VI on the block diagram.
3. Add the other objects to the block diagram as shown in the following illustration.



1.23

Numeric Constant (**Functions»Numeric**). Add three numeric constants to the block diagram. Assign the values of 32.0, 5.0, and 9.0 to the constants by using the Labeling tool.



Note: *Remember, you can use the pop up on functions and choose **Create Constant** to automatically create and wire the correct constant to a function.*



The Subtract function (**Functions»Numeric**) subtracts 32 from the Fahrenheit value for the conversion to Centigrade.



The Divide function (**Functions»Numeric**) computes the value of 5/9 for the temperature conversion.



The Multiply function (**Functions»Numeric**) returns the Centigrade value from the conversion process.

4. Wire the diagram objects as shown in the previous block diagram illustration.



Note: *A broken wire between the Thermometer icon and the Temp in deg C terminal might indicate that you have assigned the subVI connector terminal to the front panel indicator incorrectly. Review the instructions in the **Creating the SubVI** section earlier in this chapter. When you have modified the subVI, you may need to select **Relink to SubVI** from the icon pop-up menu. If necessary, choose **Edit»Remove Bad Wires**.*

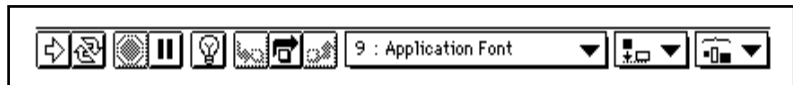


5. Return to the front panel and click on the run button in the toolbar.

Block Diagram Toolbar

The block diagram contains additional options not included on the front panel toolbar.

Block Diagram Toolbar:



The block diagram toolbar contains the following buttons that you can use for debugging VIs.



Hilite execute button—Displays data as it passes through wires



Step into button—Steps into loops, subVIs, and so on



Step over button—Begins single stepping, steps over a loop, subVI, and so on

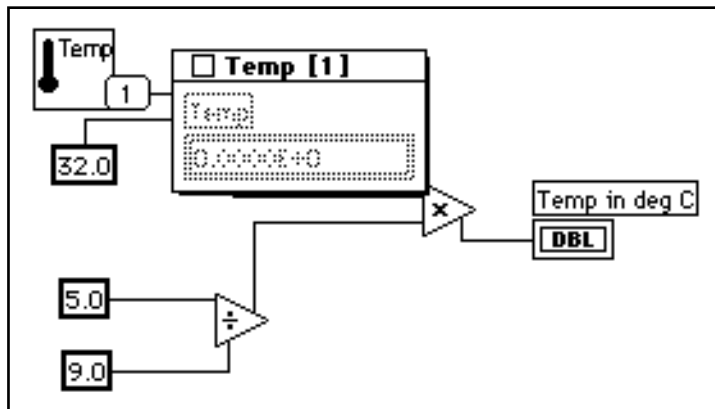


Step out button—Completes execution of loops, VIs, block diagrams, and so on

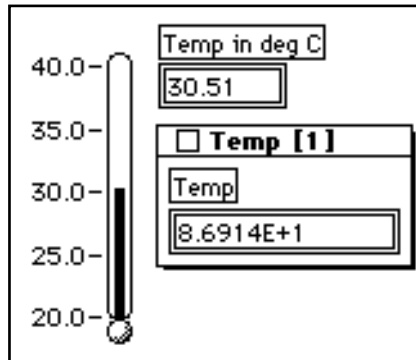
Some Debugging Techniques

The thermometer should display a value in the selected range. However, suppose you want to see the Fahrenheit value for comparison and debugging. LabVIEW contains some tools that can help you. In this exercise, you examine the probe and execution highlighting features. These techniques and other debugging tools and tips are discussed in greater detail in Chapter 9, *Programming Tips and Debugging Techniques*, of this tutorial.

1. Select **Windows»Show Diagram**.
2. Select the Probe tool from the **Tools** palette. Click with the Probe tool on the temperature value (wire) coming out of the My Thermometer subVI. A Probe window pops up with the title Temp 1 and a yellow glyph with the number of the probe, as shown in the following illustration. The Probe window also appears on the front panel.



- Return to the front panel. Move the Probe window so you can view both the probe and thermometer values as shown in the following illustration. Run the VI. The temperature in degrees Fahrenheit appears in the Probe window.

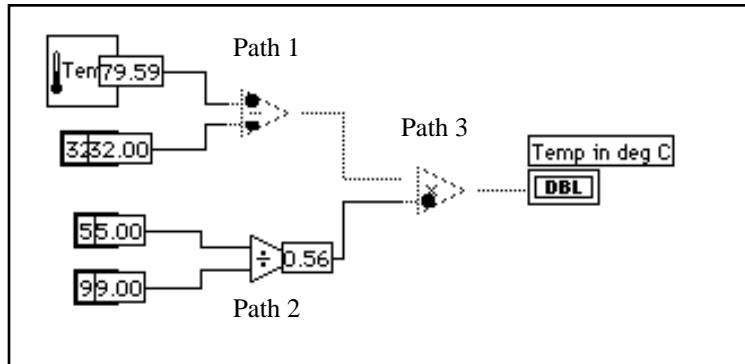


- Close the Probe window by clicking in the close box in the top-left corner of the Probe window title bar.

Another useful debugging technique is to examine the flow of data in the block diagram using LabVIEW's execution highlighting feature.



- Return to the block diagram of the VI by choosing **Windows»Show Diagram**.
- Begin execution highlighting by clicking on the hilite execute button, in the toolbar, shown at left. The hilite execute button changes to an illuminated light bulb.
- Run the VI and notice that execution highlighting animates the VI block diagram execution. Moving bubbles represent the flow of data through the VI. Also notice that data values appear on the wires and display the values contained in the wires at that time, as shown in the following block diagram, just as if you had probed the wire.



Notice the order in which the different nodes in LabVIEW execute. In conventional text-based languages, the program statements execute in the order in which they appear. LabVIEW, however, uses *data flow* programming. In data flow programming, a node executes when data is available at all of the node inputs, not necessarily in a top-to-bottom or left-to-right manner.

The preceding illustration shows that LabVIEW can multitask between paths 1 and 2 because there is no data dependency, that is, nothing in path 1 depends on data from path 2, and nothing in path 2 depends on data from path 1. Path 3 must execute last, however, because the multiply function is dependant upon the data from the Subtract and Divide functions.

Execution highlighting is a useful tool for examining the data flow nature of LabVIEW and is discussed further in Chapter 9, *Programming Tips and Debugging Techniques*, of this tutorial.

You can also use the single stepping buttons if you want to have more control over the debugging process.



8. Begin single stepping by clicking on the step over button, in the toolbar. Clicking on this button displays the first execution sequence in the VI. After LabVIEW completes this portion of the sequence, it highlights the next item that executes in the VI.



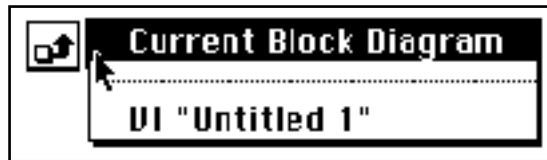
9. Step over the divide function by clicking on the step over button, in the toolbar. Clicking on this button executes the Divide function. After LabVIEW completes this portion of the sequence, it highlights the next item that executes in the VI.



10. Step into the My Thermometer subVI by clicking on the step into button, in the toolbar. Clicking on this button opens the front panel and block diagram of your thermometer subVI. You can now choose to single step through or run the subVI.



11. Finish executing the block diagram by clicking on the step out button, in the toolbar. Clicking on this button completes all remaining sequences in the block diagram. After LabVIEW completes this portion of the sequence, it highlights the next item that executes in the VI. You can also hold down the mouse button when clicking on the step out button to access a pop-up menu. On this pop-up menu, you can select how far the VI executes before pausing. The following illustration shows your finish execution options in the pop-up menu of the step out button.



12. Save the VI in mywork.llb. Name it Using My Thermometer.vi, and close the VI.

Opening, Operating, and Changing SubVIs

You can open a VI used as a subVI from the block diagram of the calling VI. You open the block diagram of the subVI by double-clicking on the subVIs icon or by selecting **Project»This VI's SubVIs**. You then open the block diagram by selecting **Windows»Show Diagram**.

Any changes you make to a subVI alter only the version in memory until you save the subVI. Notice that the changes affect all calls to the subVI and not just the node you used to open the VI.

Hierarchy Window

You use the Hierarchy window (**Project»Show VI Hierarchy**) to visually display the dependencies of VIs by providing information on VI callers and subVIs. This window contains a toolbar that you can use to configure several types of settings for displayed items. The following illustration shows an example of the VI hierarchy toolbar.



You can use buttons on the Hierarchy window toolbar or the **VIEW** menu, or pop up on an empty space in the window to access the following options.



- Redraw–Rearranges nodes after successive operations on hierarchy nodes if you need to minimize line crossings and maximize symmetric aesthetics. If a focus node exists, you then scroll through the window so that the first root that shows subVIs is visible.



- Switch to vertical layout–Arranges the nodes from top-to-bottom, placing roots at the top.



- Switch to horizontal layout–Arranges the nodes from left-to-right, placing roots on the left side.



- Include/Exclude VIs in VI libraries–Toggles the hierarchy graph to include or exclude VIs in VI libraries.



- Include/Exclude global variables–Toggles the hierarchy graph to include or exclude global variables.



- Include/Exclude typedefs–Toggles the hierarchy graph to include or exclude typedefs.

In addition, the **View** menu and pop-up menus include **Show all VIs** and **Full VI Path in Label** options that you cannot access on the toolbar.



As you move the Operating tool over objects in the Hierarchy window, LabVIEW displays the name of the VI below the VI icon.

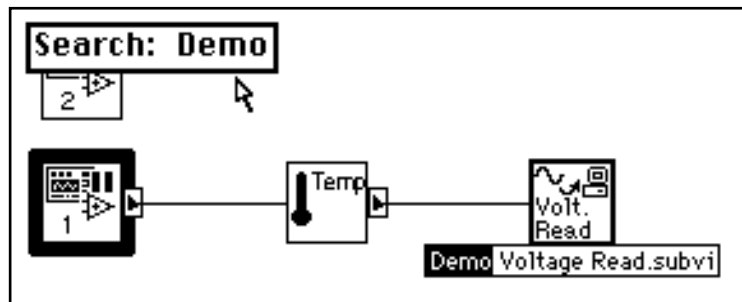
Use the <Tab> key toggle between the Positioning and Scroll window tools. This feature is useful for moving nodes from the Hierarchy window to the block diagram.

You can drag a VI or subVI node to the block diagram or copy it to the clipboard by clicking on the node. <Shift>-click on a VI or subVI node to select multiple selections for copying to other block diagrams or front panels. Double-clicking on a VI or subVI node opens the front panel of that node.

Any VIs that contain subVIs have an arrow button next to the VI that you can use to show or hide the VI's subVIs. Clicking on the red arrow button or double-clicking on the VI itself opens the VI's subVIs. A black arrow button on a VI node means that all subVIs are displayed. You can also pop up on a VI or subVI node to access a menu with options, such as showing or hiding subVIs, open the VI or subVI front panel, edit the VI icon, and so on.

Search Hierarchy

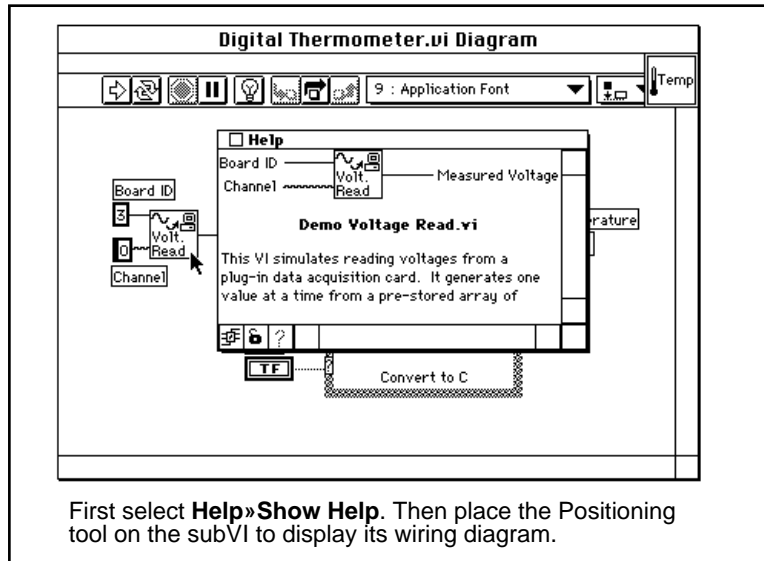
You can also search currently visible nodes in the Hierarchy window by name. You initiate the search by typing in the name of the node, anywhere on the window. As you type in the text, a search window appears, which displays the text as you type it in and concurrently searches through the hierarchy. The following illustration shows the search hierarchy.



After finding the correct node, you can press <Enter> to search for the next node that matches the search string, or you can press <Shift-Enter> (Windows); <shift-return> (Macintosh); <Shift-Return> (Sun); or <Shift-Enter> (HP-UX) to find the previous node that matches the search string.

Online Help for SubVI Nodes

When you place one of the tools on a subVI node, the Help window shows the icon for the subVI with wires attached to each terminal. The following illustration shows an example of online help. This is the Digital Thermometer VI from **Functions»Tutorial**. Your thermometer VI also contains the text you typed in the VI Information dialog box.



Simple/Complex Help View

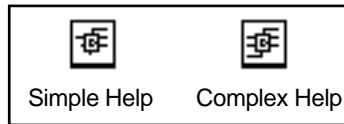
In the Help window, you can specify whether you want to display the simple or complex view for block diagram objects.



Note: *When you open the Help window, LabVIEW automatically defaults to the simple help view.*

In simple help view, LabVIEW displays only the required and recommended inputs for VIs and functions. In complex help view, LabVIEW displays the required, recommended, and optional inputs for VIs and functions. It also displays the full path name of a VI. To access the simple help view, press the Simple/Complex Diagram Help switch,

or choose **Help»Simple Diagram Help**. The following illustration shows both views of the Simple/Complex Diagram Help switch.

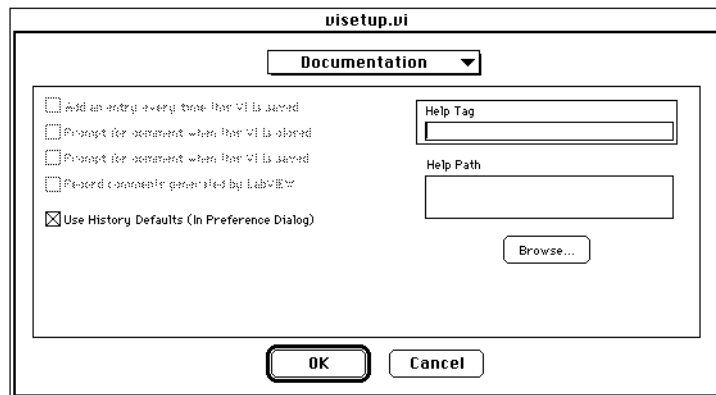


In the Help window, required inputs appear in bold text, recommended inputs appear in plain text, and optional inputs appear in gray text. When designing your own VIs, you can specify which inputs are required, recommended, or optional by popping up on an input or output on the connector pane and selecting the correct option from the **This Connection is** submenu.



Links to Online Help Files

In the Help Window, you can click on the online help button to access LabVIEW's online help as well as help files that you have created using a help compiler. If you want to create your own help file, you must specify the link to the help file by clicking on the icon pane and selecting **VI Setup....** When the VI Setup dialog box opens, choose Documentation from the ring control at the top of the box, and then enter the path of the help file in the Help Path box. The following illustration shows the options that appear in the VI Setup dialog box.



You select **Browse...** to associate the help file and topic with your VI.

For more information on creating help files, see the *Creating Your Own Help Files* section, in Chapter 25, *Managing Your Applications*, in the *LabVIEW User Manual*.

Summary

LabVIEW's ability to call VIs as subVIs within higher-level VIs facilitates modular block diagrams. Modularization, in turn, makes your block diagrams more understandable and simplifies debugging.

A VI used as a subVI must have an icon and connector. The connector terminals pass data to the subVI code and receive the results from the subVI.

You create the icon using the Icon Editor. You define the connector by choosing the number of terminals you want for the VI and then assigning a front panel control or indicator to each of those terminals.

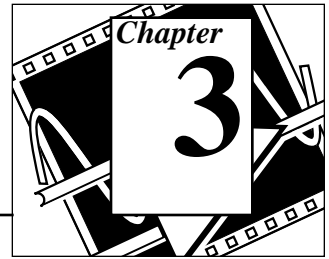
Once you have created the icon and connector for a VI, you can then use the VI as a subVI. You select subVIs using **Functions»Select a VI...**

LabVIEW contains several tools for debugging VIs. You can place probes on any wire and display the value that passes through that wire as the VI runs. Execution highlighting animates a block diagram by displaying the data flow as moving bubbles and autoprobes. You can use single stepping to debug VIs and examine data flow in VIs and subVIs. These debugging techniques and more are described further in Chapter 9, *Programming Tips and Debugging Techniques*, of this tutorial.

You use the Hierarchy window to graphically display dependencies of VIs and subVIs. With the Hierarchy window, you can choose between VI layout, including information about typedefinitions, global variables, and so on. You access the Hierarchy window by selecting **Project»Show VI Hierarchy**.

LabVIEW also includes online help for subVIs. You can use the online help to wire subVIs correctly. You can also use online help to show the simple or complex view of a VI or subVI.

Loops and Charts



You Will Learn:

- How to use a While Loop.
- How to display data in a chart.
- What a shift register is and how to use it.
- How to use a For Loop.

Structures control the flow of data in a VI. LabVIEW has four structures: the While Loop, the For Loop, the Case structure, and the Sequence structure. This chapter introduces the While Loop and For Loop structures along with the chart and the shift register. The Case and Sequence structures are explained in Chapter 5, *Case and Sequence Structures and the Formula Node*.

For examples of structures, see `examples\general\structs.llb`.

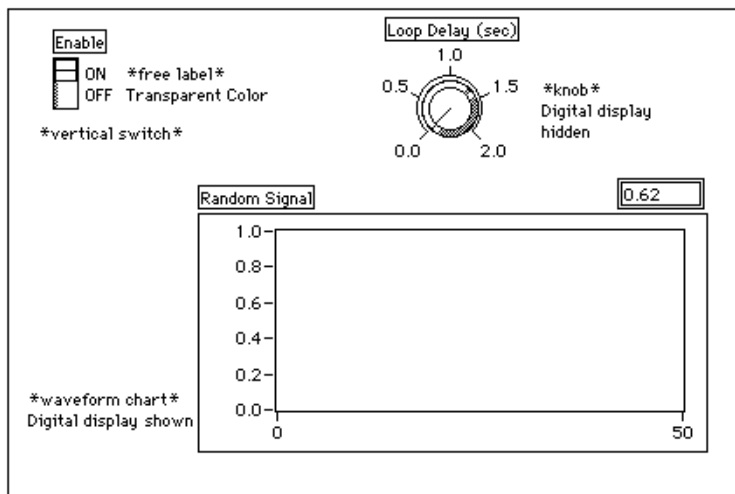
For examples of charts, see `examples\general\graphs\charts.llb`.

Using While Loops and Charts

OBJECTIVE To use a While Loop and a chart for acquiring and displaying data in real time.

You will build a VI that generates random data and displays it on a chart. A knob control on the front panel will adjust the loop rate between 0 and 2 seconds and a switch will stop the VI. You will learn to change the mechanical action of the switch so you do not have to turn on the switch each time you run the VI. Use the front panel in the following illustration to get started.

Front Panel



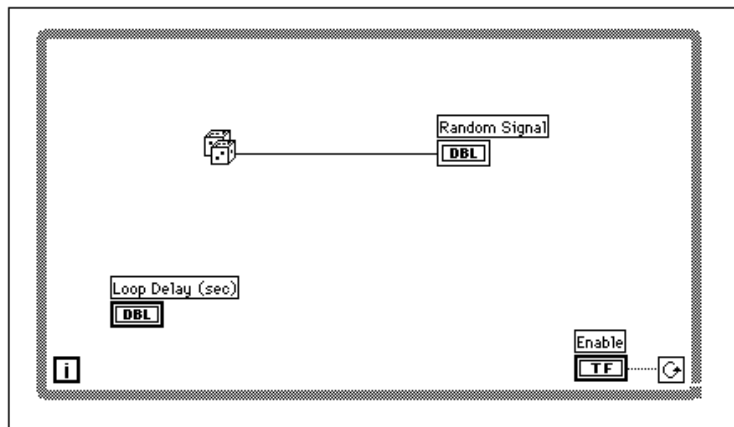
1. Open a new front panel.
2. Place a vertical switch (**Controls»Boolean**) in the front panel. Label the switch **Enable**. You use this switch to stop the acquisition.
3. Use the Labeling tool to create the free label for ON and OFF. Use the Color tool to make the free label border transparent. The T in the bottom left corner of the color palette makes an object transparent.
4. Place a waveform chart (**Controls»Graph**) in the front panel. Label the chart **Random Signal**. The chart displays random data in real time.
5. Pop up on the chart and choose **Show»Digital Display**. The digital display shows the latest value.
6. Using the Labeling tool, double-click on 10.0 in the chart, type 1.0, and click outside the label area. The click enters the value. You can also press <Enter> (Windows); <return> (Macintosh); <Return> (Sun); or <Enter> (HP-UX) to input your change to the scale.





7. Place a knob (**Controls»Numeric**) in the front panel. Label the knob Loop Delay (sec). This knob controls the timing of the While Loop later in this exercise. Pop up on the knob and deselect **Show»Digital Display** to hide the digital display that shows by default.
8. Using the Labeling tool, double-click on 10.0 in the scale around the knob, type 2.0, and click outside the label area to enter the new value.

Block Diagram



1. Open the block diagram.
2. Place the While Loop in the block diagram by selecting it from **Functions»Structures**. The While Loop is a resizable box that is not dropped on the diagram immediately. Instead, you have the chance to position and resize it. To do so, click in an area above and to the left of all the terminals. Continue holding down the mouse button, and drag out a rectangle that encompasses the terminals. A While Loop is then created with the specified location and size.

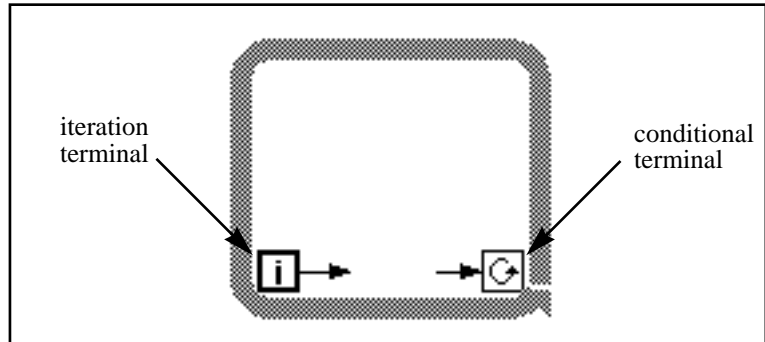
conditional terminal



iteration terminal



The While Loop, shown in the following illustration, is a resizable box you use to execute the diagram inside it until the Boolean value passed to the *conditional terminal* (an input terminal) is FALSE. The VI checks the conditional terminal at the *end* of each iteration; therefore, *the While Loop always executes at least once*. The *iteration terminal* is an output numeric terminal that contains the number of times the loop has executed. However, the iteration count always starts at zero, so if the loop runs once, the iteration terminal outputs 0.



The While Loop is equivalent to the following pseudo-code:

Do

Execute Diagram Inside the Loop (which sets the condition)

While Condition is TRUE



3. Select the Random Number (0-1) function from **Functions»Numeric**.

4. Wire the diagram as shown in the opening illustration of this *Block Diagram* section, connecting the Random Number (0-1) function to the Random Signal chart terminal, and the Enable switch to the conditional terminal of the While Loop. Leave the Loop Delay terminal unwired for now.



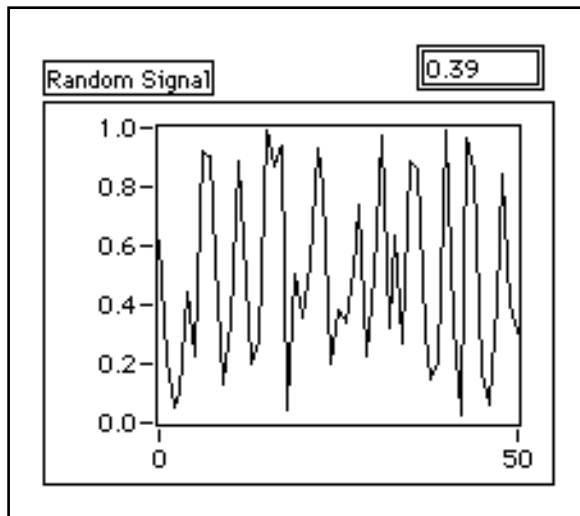
5. Return to the front panel and turn on the vertical switch by clicking on it with the Operating tool. Run the VI.

The While Loop is an indefinite looping structure. The diagram within its border executes as long as the specified condition is true. In this example, as long as the switch is on (TRUE), the diagram continues to generate random numbers and display them on the chart.

6. To stop the loop, click on the vertical switch. Turning the switch off sends the value FALSE to the loop conditional terminal and stops the loop.
7. The chart has a display buffer that retains a number of points after they have scrolled off the display. Give the chart a scrollbar by popping up on the chart and selecting **Show»Scrollbar**. You can use the Positioning tool to adjust the size and position of the scrollbar.

To scroll through the chart, click and hold down the mouse button on either arrow in the scrollbar.

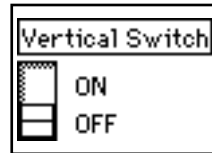
To clear the display buffer and reset the chart, pop up on the chart and choose **Data Operations»Clear Chart**.



Note: *The display buffer default size is 1,024 points. You can increase or decrease this buffer size by popping up on the chart and choosing **Chart History Length**....*

Mechanical Action of Boolean Switches

You may notice that each time you run the VI, you first must turn on the vertical switch and then click on the run button, in the toolbar. With LabVIEW, you can modify the mechanical action of Boolean controls. There are six possible choices for the mechanical action of a Boolean control—**Switch When Pressed**, **Switch When Released**, **Switch Until Released**, **Latch When Pressed**, **Latch When Released**, and **Latch Until Released**. LabVIEW contains an example that demonstrates these behaviors called *Mechanical Action of Booleans.vi* located in `examples\general\controls\booleans.llb`. As an example, consider the following vertical switch. The default value of the switch is off (FALSE).



Switch When Pressed action changes the control value each time you click on the control with the Operating tool. The action is similar to that of a ceiling light switch, and is not affected by how often the VI reads the control.



Switch When Released action changes the control value only after you release the mouse button, during a mouse click, within the graphical boundary of the control. The action is not affected by how often the VI reads the control. This action is similar to what happens when you click on a check mark in a dialog box; it becomes highlighted but does not change until you release the mouse button.



Switch Until Released action changes the control value when you click on the control. It retains the new value until you release the mouse button, at which time the control reverts to its original value. The action is similar to that of a doorbell, and is not affected by how often the VI reads the control.



Latch When Pressed action changes the control value when you click on the control. It retains the new value until the VI reads it once, at which point the control reverts to its default value. (This action happens whether or not you continue to press the mouse button.) This action is similar to that of a circuit breaker and is useful for stopping While Loops or having the VI do something only once each time you set the control.



Latch When Released action changes the control value only after you release the mouse button. When your VI reads the value once, the control reverts to the old value. This action guarantees at least one new value. As with **Switch When Released**, this action is similar to the behavior of buttons in a dialog box; clicking on this action highlights the button, and releasing the mouse button latches a reading.



Latch Until Released action changes the control value when you click on the control. It retains the value until your VI reads the value once or until you release the mouse button, depending on which one occurs last.

1. Modify the vertical switch so it is used only to stop the VI. That is, change the switch so that you need not turn on the switch each time you run the VI.
 - a. Turn on the vertical switch.
 - b. Pop up on the switch and choose **Data Operations»Make Current Value Default**. This makes the ON position the default value.
 - c. Pop up on the switch and choose **Mechanical Action»Latch When Pressed**.
2. Run the VI. Click on the vertical switch to stop the acquisition. The switch moves to the OFF position and changes back after the While Loop condition terminal reads the value.

Adding Timing

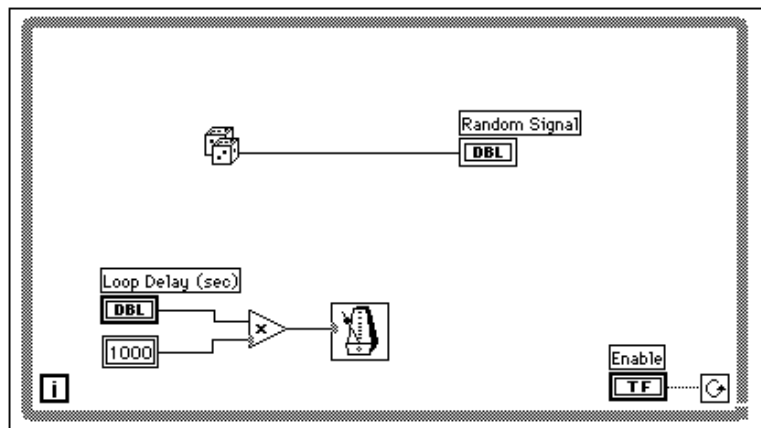
When you ran the VI, the While Loop executed as quickly as possible. However, you may want to take data at certain intervals, such as once per second or once per minute.

LabVIEW's timing functions express time in milliseconds (ms), however, your operating system may not maintain this level of timing

accuracy. The following list contains guidelines for determining the accuracy of LabVIEW's timing functions on your system.

- **(Windows 3.1)** The timer has a default resolution of 55 ms. You can configure LabVIEW to have 1 ms resolution by selecting **Edit»Preferences...**, selecting Performance and Disk from the Paths ring, and unchecking the Use Default Timer checkbox. LabVIEW does not use the 1 ms resolution by default because it places a greater load on your operating system. Read the description of the **Use Default Timer** option in the *Performance and Disk Preferences* section in Chapter 8, *Customizing Your LabVIEW Environment*, in the *LabVIEW User Manual* to decide if you should use this option.
- **(Windows 95/NT)** The timer has an resolution of 1 ms. However, this is hardware dependent, so on slower systems, such as an 80386, you may have lower resolution timing.
- **(Macintosh)** For 68K systems without the QuickTime extension, the timer has an resolution of 16 2/3 ms (1/60th of a second). If you have a Power Macintosh or have QuickTime installed, timer resolution is 1 ms.
- **(UNIX)** The timer has a resolution of 1 ms.

You can control loop timing using the Wait Until Next ms Multiple function (**Functions»Time & Dialog**). This function ensures that no iteration is shorter than the specified number of milliseconds.



1. Modify the VI to generate a new random number at a time interval specified by the knob, as shown in the preceding diagram.



Wait Until Next ms Multiple function (**Functions»Time & Dialog**). In this exercise, you multiply the knob terminal by 1000 to convert the knob value in seconds to milliseconds. Use this value as the input to the Wait Until Next ms Multiple function.



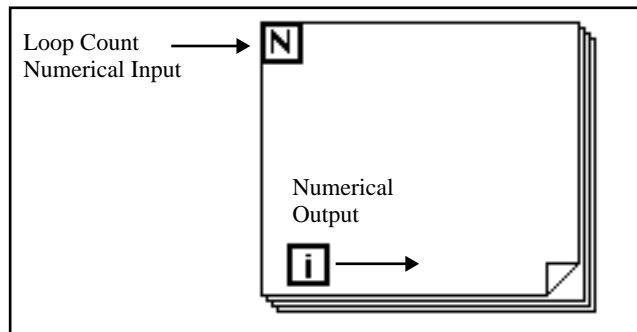
Multiply function (**Functions»Numeric**). In this exercise, the multiply function multiplies the knob value by 1000 to convert seconds to milliseconds.



Numeric Constant (**Functions»Numeric**). The numeric constant holds the constant by which you must multiply the knob value to get a quantity in milliseconds. Thus, if the knob has a value of 1.0, the loop executes once every 1000 milliseconds (once a second).

2. Run the VI. Rotate the knob to get different values for the number of seconds.
3. Save and close the VI in mywork.11b. Name it My Random Signal.vi.

For Loop



You place the *For Loop* on the block diagram by selecting it from **Functions»Structures**. A For Loop (see preceding illustration) is a resizable box, like the While Loop. Like the While Loop, it is not dropped on the diagram immediately. Instead, a small icon representing the For Loop appears in the block diagram, and you have the opportunity to size and position it. To do so, first click in an area above and to the left of all the terminals. While holding down the mouse button, drag out a rectangle that encompasses the terminals you want to place inside the For Loop. When you release the mouse button,

LabVIEW creates a For Loop of the correct size and in the position you selected.

The For Loop executes the diagram inside its border a predetermined number of times. The For Loop has two terminals:



the *count terminal* (an input terminal) The count terminal specifies the number of times to execute the loop.



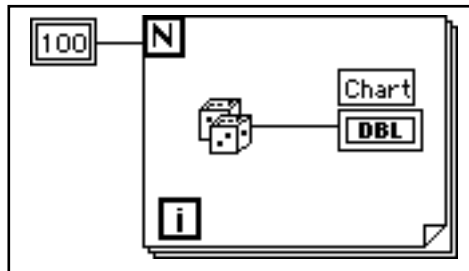
the *iteration terminal* (an output terminal). The iteration terminal contains the number of times the loop has executed.

The For Loop is equivalent to the following pseudo-code:

```
For i = 0 to N-1
```

```
Execute Diagram Inside The Loop
```

The example in the following illustration shows a For Loop that generates 100 random numbers and displays the points on a chart.



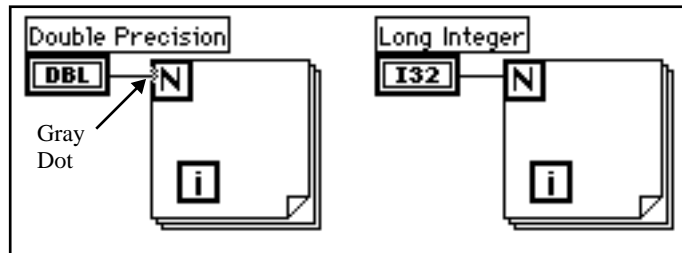
Numeric Conversion

Until now, all the numeric controls and indicators that you have used have been double-precision, floating-point numbers. LabVIEW, however, can represent numerics as integers (byte, word, or long) or floating-point numbers (single-, double-, or extended-precision). The default representation for a numeric is a double-precision, floating-point.

If you wire two terminals together that are of different data types, LabVIEW converts one of the terminals to the same representation as the other terminal. As a reminder, LabVIEW places a gray dot, called a coercion dot, on the terminal where the conversion takes place.



For example, consider the For Loop count terminal. The terminal representation is a long integer. If you wire a double-precision, floating-point number to the count terminal, LabVIEW converts the number to a long integer. Notice the gray dot in the count terminal of the first For Loop.



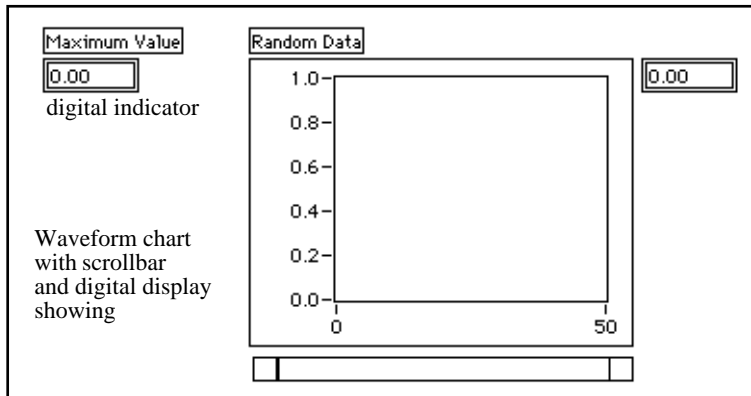
Note:

When the VI converts floating-point numbers to integers, it rounds to the nearest integer. If a number is exactly halfway between two integers, it is rounded to the nearest even integer. For example, the VI rounds 6.5 to 6, but rounds 7.5 to 8. This is an IEEE Standard method for reading numbers. See the IEEE Standard 754 for details.

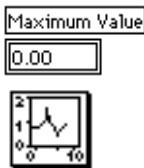
Using a For Loop

OBJECTIVE To use a For Loop and shift registers to calculate the maximum value in a series of random numbers. You will use a For Loop (N = 100) instead of a While Loop.

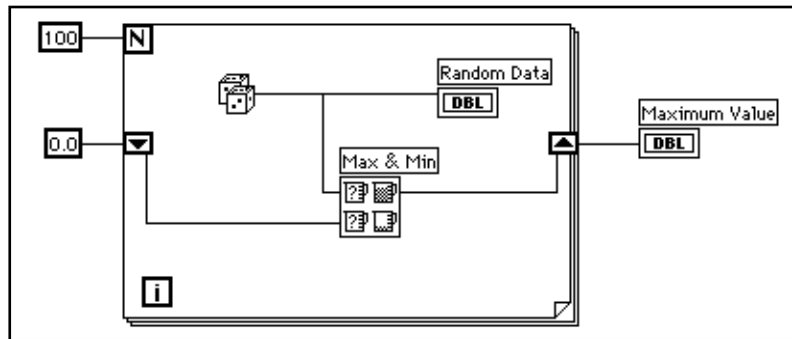
Front Panel



1. Open a new front panel and add the objects shown in the preceding illustration to it.
 - a. Place a digital indicator on the front panel and label it Maximum Value.
 - b. Place a waveform chart on the front panel and name it Random Data. Change the scale of the chart to range from 0.0 to 1.0.
 - c. Pop up on the chart and choose **Show»Scrollbar** and **Show»Digital Display**.



Block Diagram



1. Open the block diagram.
2. Add the **For Loop (Functions»Structures)**.
3. Add the shift register by popping up on the right or left border of the For Loop and choosing **Add Shift Register**.
4. Add the other objects to the block diagram.



Random Number (0-1) function (**Functions»Numeric**) to generate the random data.



Numeric Constant (**Functions»Numeric**). The For Loop needs to know how many iterations to make. In this case, you execute the For Loop 100 times.



Numeric Constant (**Functions»Numeric**). You set the initial value of the shift register to zero for this exercise because you know that the output of the random number generator is from 0.0 to 1.0.

You must know something about the data you are collecting to initialize a shift register. For example, if you initialize the shift register to 1.0, then that value is already greater than all the expected data values, and is always the maximum value. If you did not initialize the shift register, then it would contain the maximum value of a previous run of the VI. Therefore, you could get a maximum output value that is not related to the current set of collected data.



Max & Min function (**Functions»Comparison**) takes two numeric inputs and outputs the maximum value of the two in the top right corner and the minimum of the two in the bottom right corner. Because you are only interested in the maximum value for this exercise, wire only the maximum output and ignore the minimum output.

5. Wire the terminals as shown. If the Maximum Value terminal was inside the For Loop, you would see it continuously updated, but because it is outside the loop, it contains only the last calculated maximum.



Note: *Updating indicators each time a loop iterates is time-consuming and you should try to avoid it when possible to increase execution speed.*

6. Run the VI.
7. Save the VI. Name the VI My Calculate Max.vi.

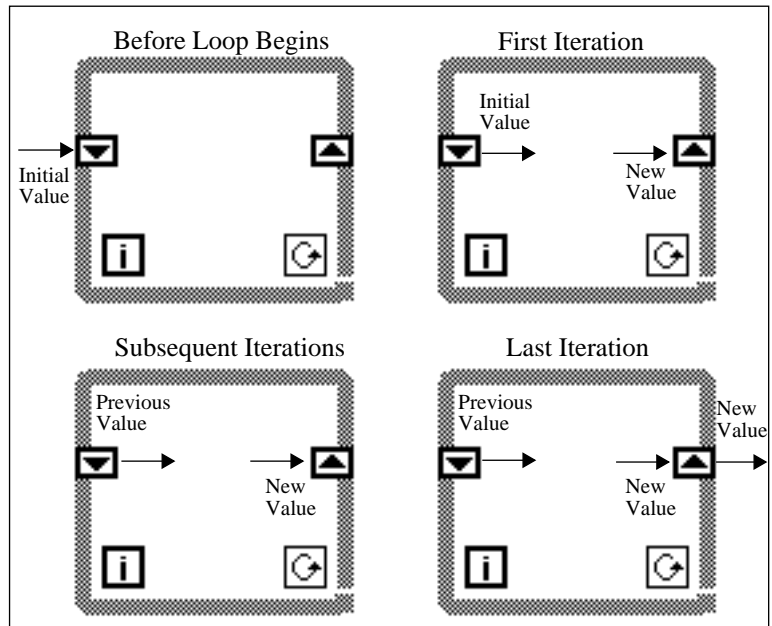
Shift Registers

Shift registers (available for While Loops and For Loops) transfer values from one loop iteration to the next. You create a shift register by popping up on the left or right border of a loop and selecting **Add Shift Register**.

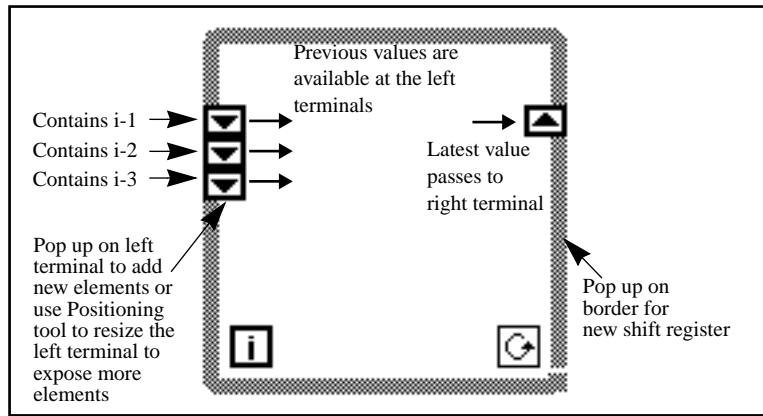


The shift register contains a pair of terminals directly opposite each other on the vertical sides of the loop border. The *right* terminal stores the data upon the completion of an iteration. That data shifts at the end of the iteration and appears in the *left* terminal at the beginning of the

next iteration (see the following illustration). A shift register can hold any data type—numeric, Boolean, string, array, and so on. The shift register automatically adapts to the data type of the first object that you wire to the shift register.



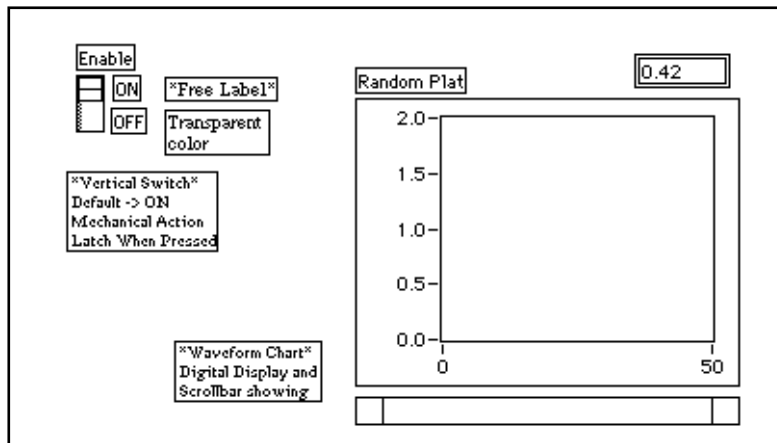
You can configure the shift register to remember values from several previous iterations. This feature is useful for averaging data points. You create additional terminals to access values from previous iterations by popping up on the *left* or *right* terminal and choosing **Add Element**. For example, if a shift register contains three elements in the left terminal, you can access values from the last three iterations.



Using Shift Registers

OBJECTIVE You will build a VI that displays two random plots on a chart. The two plots should consist of a random plot and a running average of the last four points of the random plot.

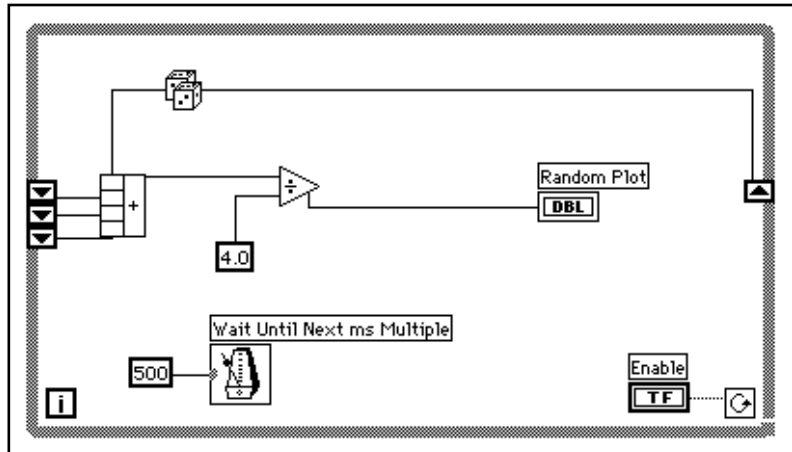
Front Panel



1. Open a new front panel and create the front panel shown in the preceding illustration.

2. After you add the waveform chart to the front panel, change the scale to range from 0.0 to 2.0.
3. After adding the vertical switch, set the ON state to be the default and set the mechanical action to **Latch When Pressed**.

Block Diagram



1. Add the **While Loop (Functions»Structures)** in the block diagram and create the shift register.
 - a. Pop up on the left or right border of the While Loop and choose **Add Shift Register**.
 - b. Add an extra element by popping up on the *left* terminal of the shift register and choosing **Add Element**. Add a third element in the same manner as the second.
2. Build the block diagram shown in the previous illustration.



Random Number (0-1) function (**Functions»Numeric**) generates raw data.

Compound Arithmetic function (**Functions»Numeric**). In this exercise, the compound arithmetic function returns the sum of random numbers from two iterations. To add more inputs, pop up on an input and choose **Add Input** from the pop-up menu.



Divide function (**Functions»Numeric**). In this exercise, the divide function returns the average of the last four random numbers.



Numeric Constant (**Functions»Numeric**). During each iteration of the While Loop, the Random Number (0-1) function generates one random value. The VI adds this value to the last three values stored in the left terminals of the shift register. The Random Number (0-1) function divides the result by four to find the average of the values (the current value plus the previous three). The average is then displayed on the waveform chart.



Wait Until Next ms Multiple function (**Functions»Time & Dialog**), ensures that each iteration of the loop occurs no faster than the millisecond input. The input is 500 milliseconds for this exercise. If you pop up on the icon and choose **Show»Label**, the label Wait Until Next ms Multiple appears.

3. Pop up on the input of the Wait Until Next ms Multiple function and select **Create Constant**. A numeric constant appears and is automatically wired to the function.
4. Use the Labeling tool to enter 500. The numeric constant wired to the Wait Until Next ms Multiple function specifies a wait of 500 milliseconds (one half-second). Thus, the loop executes once every half-second.



Notice that the VI initializes the shift registers with a random number. If you do not initialize a shift register terminal, it contains the default value or the last value from the previous run. In this case, the first few averages would be meaningless.

5. Run the VI and observe the operation. LabVIEW only plots the average on the graph.

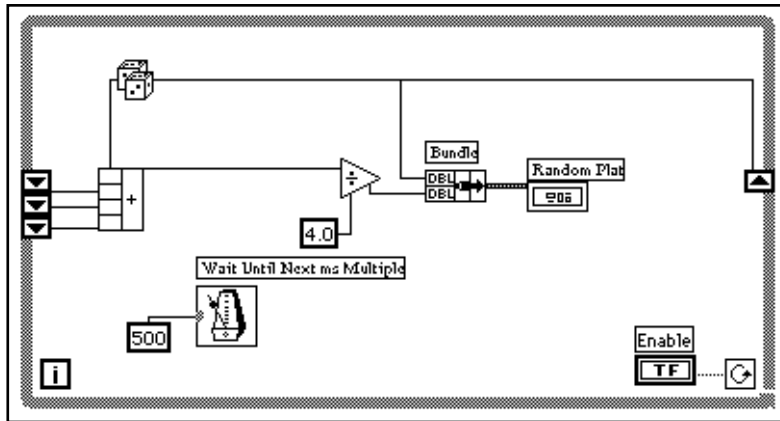


Note: *Remember to initialize shift registers to avoid incorporating old or default data into your current data measurements*

Multiplot Charts

Charts can accommodate more than one plot. You must bundle the data together in the case of multiple scalar inputs.

You should modify the block diagram to display both the average and the current random number on the same chart.



1. Modify the block diagram as shown in the previous illustration.

Bundle function (**Functions»Cluster**). In this exercise, the Bundle function *bundles*, or groups, the average and current value for plotting on the chart. The bundle node appears as shown at left when you place it in the block diagram. If you pop up on the bundle and choose **Show»Label**, the word `Bundle` appears in the label. You can add additional elements by using the Resizing cursor (accessed by placing the Positioning tool at the corner of the function) to enlarge the node.



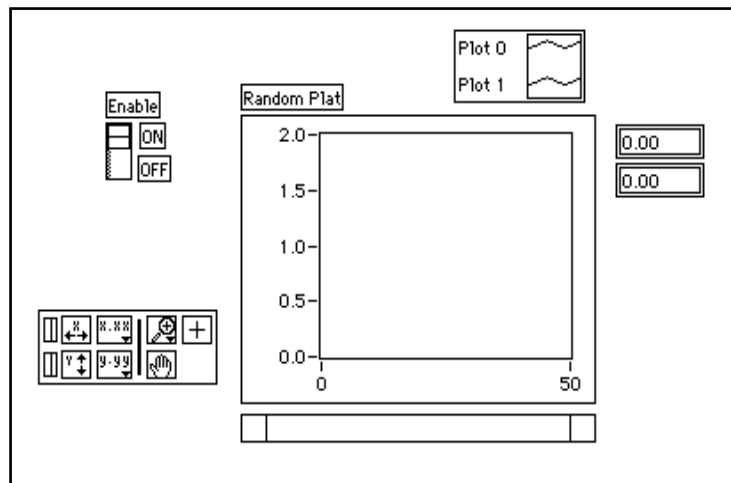
Note:

The order of the inputs to the Bundle function determines the order of the plots on the chart. For example, if you wire the raw data to the top input of the Bundle and the average to the bottom, the first plot corresponds to the raw data and the second plot corresponds to the average.

2. Run the VI. The VI displays two plots on the chart. The plots are overlaid. That is, they share the same vertical scale. Try running the VI with execution highlighting turned on to see the data in the shift registers. Remember to turn off the hilite execute button, in the toolbar, when you finish so the VI can execute at full speed.

Customizing Charts

You can customize charts to match your data display requirements or to display more information. Features available for charts include: a scrollbar, a legend, a palette, and a digital display.



On the chart, the digital display has been enabled. Notice that a separate digital display exists for each trace on the chart.

1. If the scrollbar is present, hide it by popping up on the chart and deselecting **Show»ScrollBar**.
2. Customize the Y axis.



- a. Use the Labeling tool to double-click on 2.0 in the Y scale. Type in 1.2 and press <Enter> (Windows); <return> (Macintosh); <Return> (Sun); or <Enter> (HP-UX).
- b. Again using the Labeling tool, click on the second number from the bottom on the Y axis. Change this number to 0.2, 0.5, or something other than the current number. This number determines the numerical spacing of the Y axis divisions.



Note: *The chart size has a direct effect on the display of axis scales. Increase the chart size if you have trouble customizing the axis.*

3. Show the legend by popping up on the chart, and choosing **Show»Legend**. Move the legend if necessary.



You can place the legend anywhere relative to the chart. Stretch the legend to include two plots using the Resizing cursor. The Positioning tool changes to the Resizing cursor to indicate that you can resize the legend. Rename 0 to **Current Value** by double-clicking on the label with the Labeling tool and typing in the new text. You can change plot 1 to **Running Avg** in the same way. If the text disappears, enlarge the legend text box by resizing from the *left* corner of the legend with the Resizing cursor. You can set the plot line style and the point style by popping up on the plot in the legend.

You can set the plot line width by popping up on the plot in the legend. Using this pop-up menu, you can change the default line setting to one that is larger than 1 pixel. You can also select a hairline width, which is not displayed on the computer screen, but is printed if your printer supports hairline printing.



If you have a color monitor, you can also color the plot background, traces, or point style by popping up on what you want to change in the legend with the Color tool. Choose the color you want from the color palette that appears.

4. Show the chart pop-up palette by popping up on the chart and choosing **Show»Palette**.

With the palette, you can modify the chart display while the VI is running. You can reset the chart, scale the X or Y axis, and change the display format at any time. You can also scroll to view other areas or zoom into areas of a graph or chart. Like the legend, you can place the palette anywhere relative to the chart.

5. Run the VI. While the VI is running, use the buttons from the palette to modify the chart.



You can use the X and Y buttons to rescale the X and Y axes, respectively. If you want the graph to autoscale either of the scales continuously, click on the lock switch to the left of each button to lock on autoscaling.



You can use the other buttons to modify the axis text precision or to control the operation mode for the chart. Experiment with these buttons to explore their operation, scroll the area displayed, or zoom in on areas of the chart.

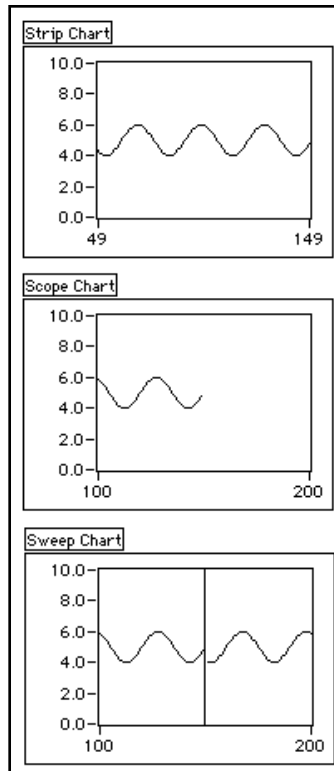


Note:

Modifying the axis text format often requires more physical space than was originally set aside for the axis. If you change the axis, the text may become larger than the maximum size that the waveform can correctly present. To correct this, use the Resizing cursor to make the display area of the chart smaller.

Different Chart Modes

The following illustration shows the three chart display options available from the **Data Operations»Update Mode**: strip chart, scope chart, and sweep chart. The default mode is strip chart. (If the VI is still running, the **Data Operations** submenu is the pop-up menu for the chart.)



The *strip chart* mode scrolling display is similar to a paper tape strip chart recorder. As the VI receives each new value, it plots the value at the right margin, and shifts old values to the left.

1. Make sure the VI is still running, pop up on the chart, and select **Data Operations»Update Mode»Scope Chart**.

The *scope chart* mode has a retracing display similar to an oscilloscope. As the VI receives each new value, it plots the value to the right of the last value. When the plot reaches the right border of the plotting area, the VI erases the plot and begins plotting again from the left border. The scope chart is significantly faster than the strip chart because it is free of the overhead processing involved in scrolling.

2. Make sure the VI is still running, pop up on the chart, and select **Data Operations»Update Mode»Sweep Chart**.

The *sweep chart* mode acts much like the scope chart, but it does not go blank when the data hits the right border. Instead, a moving vertical line marks the beginning of new data and moves across the display as the VI adds new data.

3. Stop the VI, and save it. Name it My Random Average.vi.

Summary

LabVIEW has two structures to repeat execution of a subdiagram—the While Loop and the For Loop. Both structures are resizable boxes. You place the subdiagram to be repeated inside the border of the loop structure. The While Loop executes as long as the value at the conditional terminal is TRUE. The For Loop executes a set number of times.

You can control the loop timing by using the Wait Until Next ms Multiple function. This function ensures that no iteration is shorter than a specified number of milliseconds (1000 ms equals one second).

Shift registers (available for While Loops and For Loops) transfer values from one iteration to the beginning of the next. You can configure shift registers to access values from previous iterations. For each iteration you want to recall, you must add a new element to the left terminal of the shift register.

When LabVIEW must force the numeric representation of one terminal to match the numeric representation of another terminal, a gray coercion dot appears. This dot is located at the terminal where the VI converts the data.

Additional Topics

The rest of this chapter discusses more advanced topics. Feel free to explore this material now, or to go on to the next chapter and refer back to these topics as necessary.

Customizing Charts

For more information on charts, refer to Chapter 15, *Graph and Chart Controls and Indicators*, in your *LabVIEW User Manual*.

Faster Chart Updates

You can pass an array of multiple values to the chart. The chart treats these inputs as new data for a single plot. Refer to the `charts.vi` example located in `examples\general\graphs\charts.llb`.

Stacked Versus Overlaid Plots

Earlier in this chapter you made a multiplot chart that had the plots overlaid. You can also stack plots on a chart. Refer to the `charts.vi` example located in `examples\general\graphs\charts.llb`

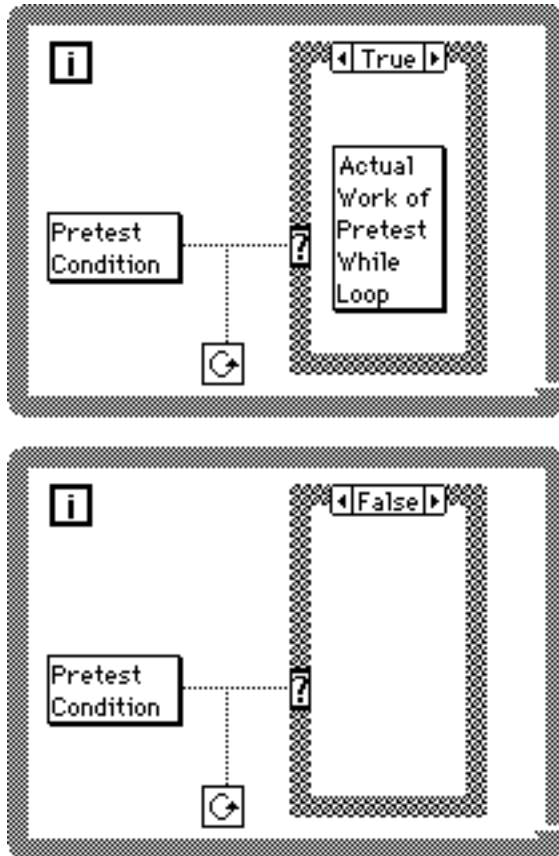
Using Loops

While and For Loops are basic structures for programming with LabVIEW, so you can find them in most of the LabVIEW examples as well as the exercises in this tutorial. You can also find more information on loops in Chapter 19, *Structures*, in the *LabVIEW User Manual*.

Testing a While Loop before Execution

The While Loop always executes at least once, because LabVIEW performs the loop test for continuation after the diagram executes. You can construct a While Loop that pretests its conditional terminal by including a Case structure inside the loop. You wire a Boolean input to the Case structure selector terminal so the subdiagram for the FALSE condition executes if the While Loop is not supposed to execute. The

subdiagram for the TRUE condition contains the work of the While Loop. The test for continuation occurs outside the Case structure, and its results are wired to both the conditional terminal of the While Loop and the selector terminal of the Case structure. In the following illustration, labels represent the pretest condition and the actual work performed by the While Loop.



This example has the same result as the following pseudocode.

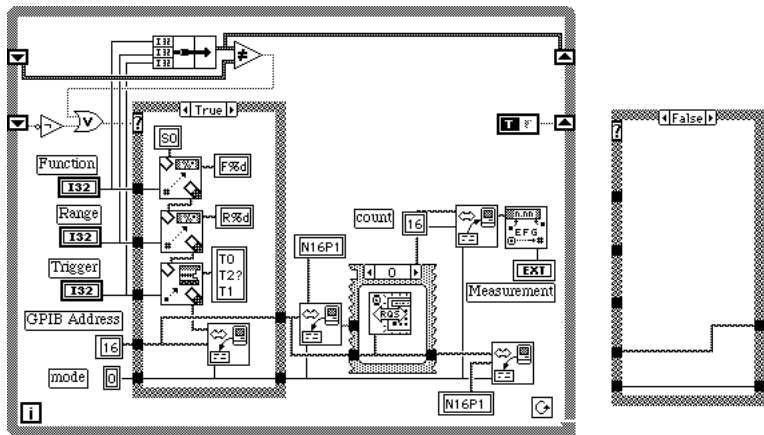
```
While (pretest condition)
Do actual work of While Loop
Loop
```

Using Uninitialized Shift Registers

You initialize a shift register by wiring a value from outside a While or For Loop to the left terminal of the shift register. Sometimes, however, you want to repeatedly execute a VI with a loop and shift register, so that each time the VI executes, the initial output of the shift register is the last value from the previous execution. To do that, you must leave the left, shift register terminal unwired from outside the loop.

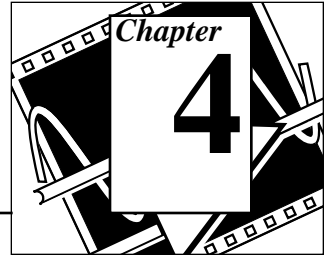
You can use uninitialized shift registers, for example, to avoid reprogramming the function, range, and trigger parameters in instrument driver VIs every time they execute. This can improve performance in instruments slow to execute commands.

The following version of a Fluke 8840A multimeter driver uses two uninitialized shift registers to remember the last state of the driver so that you have to reprogram the instrument only when you first use it or when a control parameter changes.



The first time this VI executes after you load or compile it, the value of the bottom shift register is FALSE, the default for an uninitialized Boolean. The True case executes and programs the function, range, and trigger parameters and sets the meter to use slow measurements. The True case also executes if the current value of any of the three parameter controls differs from the last value; that is, when any of the controls changes. You can modify the operation to program the changed control only by using separate Case structures for each control.

Case structures are discussed in greater detail in Chapter 5, *Case and Sequence Structures and the Formula Node*.



Arrays, Clusters, and Graphs

You Will Learn:

- About arrays.
- How to generate arrays on loop boundaries.
- What polymorphism is.
- About clusters.
- How to use graphs to display data.
- How to use some basic array functions.

Arrays

An array consists of a collection of data elements that are all the same type. An array has one or more dimensions and up to $2^{31} - 1$ elements per dimension, memory permitting. Arrays in LabVIEW can be any type (except array, chart, or graph). You access each array element through its index. The index is in the range 0 to $n-1$, where n is the number of elements in the array. The following one-dimensional array of numeric values illustrates this structure. Notice that the first element has index 0, the second element has index 1, and so on.

index	0	1	2	3	4	5	6	7	8	9
10-element array	1.2	3.2	8.2	8.0	4.8	5.1	6.0	1.0	2.5	1.7

Array Controls, Constants, and Indicators

You create array controls, constants, and indicators on the front panel or block diagram by combining an *array constant* with a numeric, Boolean, string, or cluster. The array element cannot be another array, chart, or graph.

For examples of arrays, see `examples\general\arrays.llb`.

Graphs

A *graph indicator* consists of a two-dimensional display of one or more data arrays called *plots*. LabVIEW has three types of graphs: *XY graphs*, *waveform graphs*, and *intensity graphs* (see the *Additional Topics* section at the end of this chapter for information on intensity graphs).

The difference between a graph and a chart (discussed in Chapter 3, *Loops and Charts*, in this tutorial) is that a graph plots data as a block, whereas a chart plots data point by point or array by array.

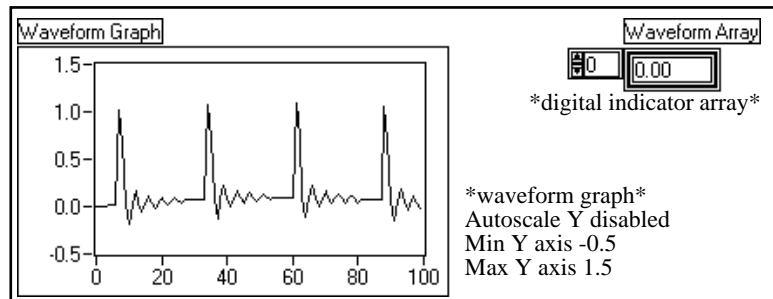
For examples of graph VIs, see `examples\general\graphs`.

Creating an Array with Auto-Indexing

OBJECTIVE To create an array using the auto-indexing feature of a For Loop and plot the array in a waveform graph.

You will build a VI that generates an array using the Generate Waveform VI and plots the array in a waveform graph. You will also modify the VI to graph multiple plots.

Front Panel

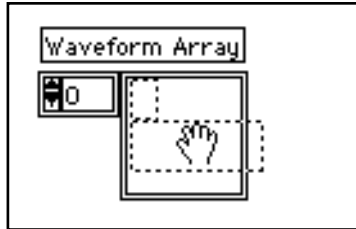


1. Open a new front panel.
2. Place an array shell from **Controls»Array & Cluster** in the front panel. Label the array constant `Waveform Array`.



1.23

- Place a digital indicator from **Controls»Numeric** inside the element display of the array constant, as the following illustration shows. This indicator displays the array contents.



As stated previously, a *graph indicator* is a two-dimensional display of one or more data arrays called *plots*. LabVIEW has three types of graphs: *XY graphs*, *waveform graphs*, and *intensity graphs*.

- Place a waveform graph from **Controls»Graph** in the front panel. Label the graph `Waveform Graph`.

The waveform graph plots arrays with uniformly spaced points, such as acquired time-varying waveforms.



- Enlarge the graph by dragging a corner with the Resizing cursor.

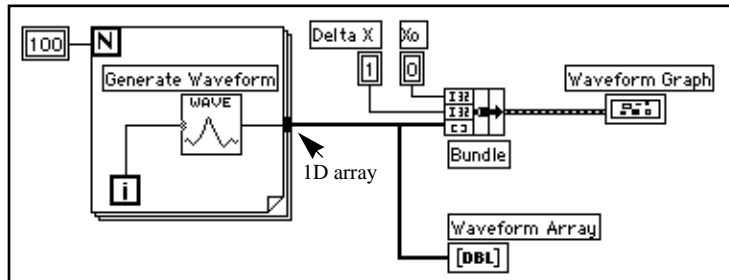
By default, graphs *autoscale* their input. That is, they automatically adjust the X and Y axis scale limits to display the entire input data set.

- Disable autoscaling by popping up on the graph and deselecting **Y Scale»Autoscale Y**.



- Modify the Y axis limits by double-clicking on the scale limits with the Labeling tool and entering the new numbers. Change the Y axis minimum to -0.5 and the maximum to 1.5.

Block Diagram

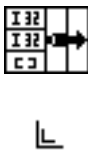


1. Build the block diagram shown in the preceding illustration.

The Generate Waveform VI (**Functions»Tutorial**) returns one point of a waveform. The VI requires a scalar index input, so wire the loop iteration terminal to this input. Popping up on the VI and selecting **Show»Label** displays the word Generate Waveform in the label.

Notice that the wire from the Generate Waveform VI becomes thicker as it changes to an array at the loop border.

The For Loop automatically accumulates the arrays at its boundary. This is called *auto-indexing*. In this case, the numeric constant wired to the loop count numeric input has the For Loop create a 100-element array (indexed 0 to 99).



Bundle function (**Functions»Cluster**) assembles the plot components into a cluster. You need to resize the Bundle function icon before you can wire it properly. Place the Positioning tool on the lower right corner of the icon. The tool transforms into the Resizing cursor shown at left. When the tool changes, click and drag down until a third input terminal appears. Now, you can continue wiring your block diagram as shown in the first illustration in this section.

A cluster consists of a data type that can contain data elements of different types. The cluster in the block diagram you are building here groups related data elements from multiple places on the diagram, reducing wire clutter. When you use clusters, your subVIs require fewer connection terminals. A cluster is analogous to a record in Pascal

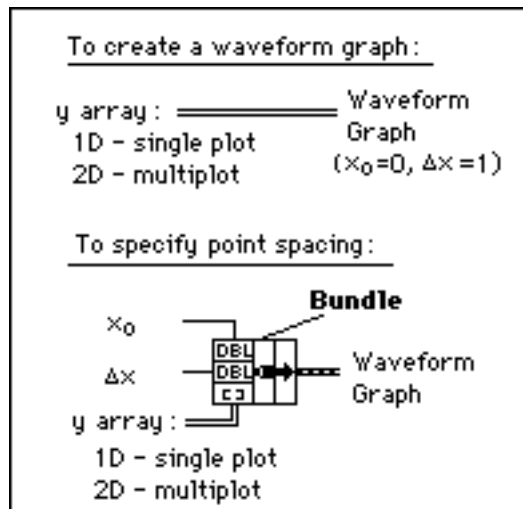
or a struct in C. You can think of a cluster as a bundle of wires, much like a telephone cable. Each wire in the cable would represent a different element of the cluster. The components include the initial X value (0), the delta X value (1), and the Y array (waveform data, provided by the numeric constants on the block diagram). In LabVIEW, use the Bundle function to assemble a cluster.



Note: *Be sure to build data types that the graphs and charts accept.*

As you build your block diagram, be sure to check your data types by taking the following steps:

- Open the Help window by choosing **Help»Show Help**.
- Move the Wiring tool over the graph terminal.
- Check the data type information that appears in the Help window. For an example, see the following illustration.



Numeric Constant (**Functions»Numeric**). Three numeric constants set the number of For Loop iterations, the initial X value, and the delta X value. Notice that you can pop up on the For Loop count terminal, shown at left, and select **Create Constant** to automatically add and wire a numeric constant for that terminal.



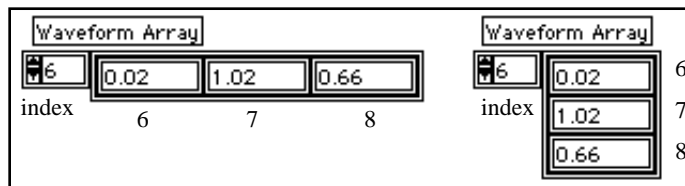
Each iteration of the For Loop generates one point in a waveform that the VI stores in the waveform array created automatically at the loop border. After the loop finishes execution, the Bundle function bundles the initial value of X (X_0), the delta value of X, and the array for plotting on the graph.

2. Return to the front panel and run the VI. The VI plots the auto-indexed waveform array on the waveform graph. The initial X value is 0 and the delta X value is 1.
3. Change the delta X value to 0.5 and the initial X value to 20. Run the VI again.

Notice that the graph now displays the same 100 points of data with a starting value of 20 and a delta X of 0.5 for each point (see the X axis). In a timed test, this graph would correspond to 50 seconds worth of data starting at 20 seconds. Experiment with several combinations for the initial and delta X values.

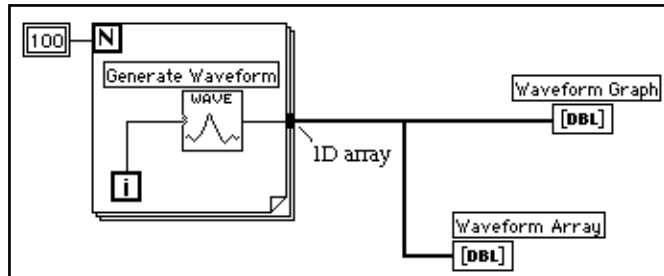
4. You can view any element in the array by entering the index of that element in the index display. If you enter a number greater than the array size, the display dims, indicating that you have not defined a value for that index.

If you want to view more than one element at a time, you can resize the array indicator. Place the Positioning tool on the lower right corner of the array. The tool transforms into the Resizing cursor shown at left. When the tool changes, drag to the right or straight down. The array now displays several elements in ascending index order, beginning with the element corresponding to the specified index, as the following illustration shows.



In the previous block diagram, you specified an initial X and a delta X value for the waveform. Often, however, the initial X value is zero and the delta X value is 1. In these instances, you can wire the waveform

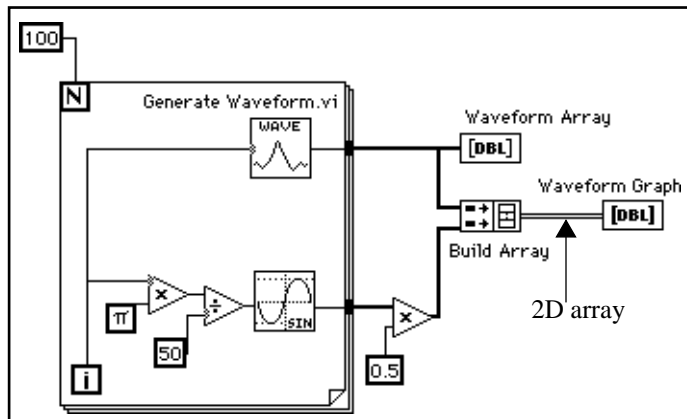
array directly to the waveform graph terminal, as the following illustration shows.



- Return to the block diagram. Delete the Bundle function and the numeric constants wired to it. To delete the function and constants, select the function and constants with the Positioning tool then press <Delete>. Select **Edit>Remove Bad Wires**. Finish wiring the block diagram as shown in the previous illustration.
- Run the VI. Notice that the VI plots the waveform with an initial X value of 0 and a delta X value of 1.

Multiplot Graphs

You can create multiplot waveform graphs by building an array of the data type normally passed to a single-plot graph.



1. Continue building your block diagram as shown in the preceding diagram.



Sine function from (**Functions»Numeric»Trigonometric**). In this exercise, you use the function in a For Loop to build an array of points that represents one cycle of a sine wave.



Build Array function (**Functions»Array**). In this exercise, you use this function to create the proper data structure to plot two arrays on a waveform graph, which in this case is a two-dimensional array. Enlarge the Build Array function to create two inputs by dragging a corner with the Positioning tool.



Pi constant (**Functions»Numeric»Additional Numeric Constants**).

Remember that you can find the Multiply and Divide functions in **Functions»Numeric**.

2. Switch to the front panel. Run the VI.

Notice that the two waveforms plot on the same waveform graph. The initial X value defaults to 0 and the delta X value defaults to 1 for both data sets.



Note:

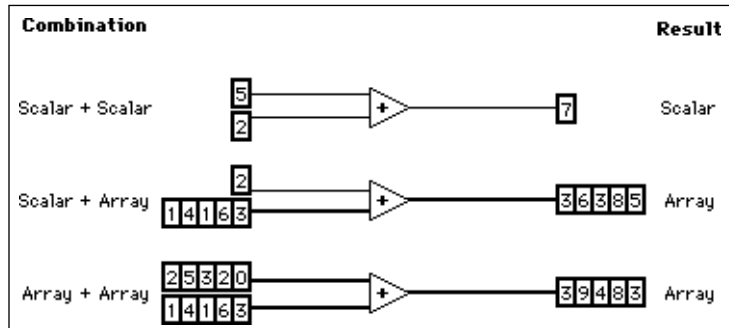
*You can change the appearance of a plot on the graph by popping up in the legend for a particular plot. For example, you can change from a line graph to a bar graph by choosing **Common Plots»Bar Graph**.*

3. Save and close the VI. Name it My Graph Waveform Arrays.vi. Be sure to save your work in mywork.llb.

Polymorphism

Polymorphism is the ability of a function to adjust to input data of different types, dimensions, or representations. Most LabVIEW functions are polymorphic. The previous block diagram is an example of polymorphism. Notice that you use the Multiply function in two locations, inside and outside the For Loop. Inside the For Loop, the function multiplies two scalar values; outside the For Loop, the function multiplies an array by a scalar value.

The following example shows some of the polymorphic combinations of the Add function.



In the first combination, the two scalars are added together, and the result is a scalar. In the second combination, the scalar is added to each element of the array, and the result is an array. In the third combination, each element of one array is added to the corresponding element of the other array. You can also use other combinations, such as clusters of numerics, arrays of clusters, and so on.

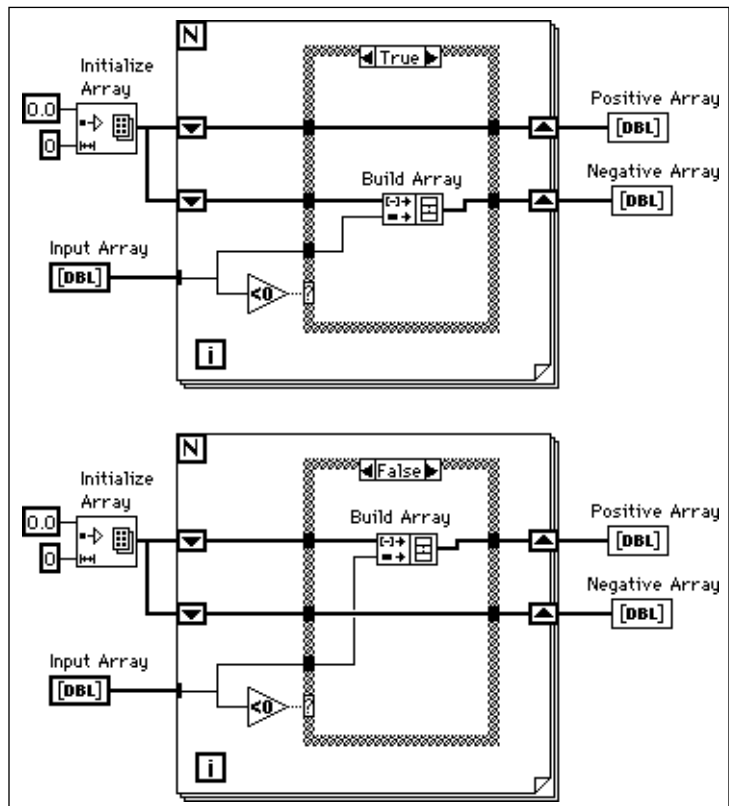
These principles can be applied to other LabVIEW functions and data types. LabVIEW functions may be polymorphic to different degrees. Some functions may accept numeric and Boolean inputs, others may accept a combination of any data types. For more information about polymorphism, see [Online Reference»Function and VI Reference»Introduction to Functions](#).

Using Auto-Indexing on Input Arrays

OBJECTIVE You will open and operate a VI that uses auto-indexing in a For Loop to process an array.

1. Open the `Separate Array Values.vi` by selecting **File»Open...** The VI is located in `examples\general\arrays.llb`.
2. Open the block diagram. You can pop up on the array and choose **Show Case True** or **Show Case False** to view the true and false cases of the array.

The following illustration shows the block diagram with both True and False cases visible.



Notice that the wire from Input Array changes from a thick wire outside the For Loop, indicating it is an array, to a thin wire inside the loop, indicating it is a single element. The i^{th} element of the array is automatically indexed from the array during each iteration.

Using Auto-Indexing to Set the For Loop Count

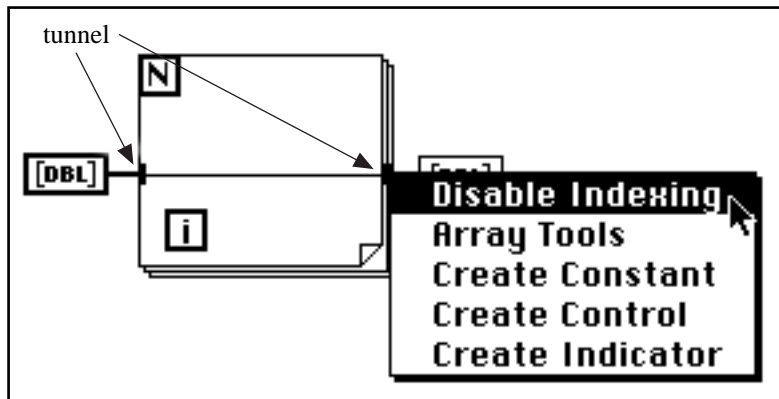
Notice that the count terminal is left unwired. When you use auto-indexing on an array *entering* a For Loop, LabVIEW automatically sets the count to the array size, eliminating the need to wire a value to the count terminal. If you use auto-indexing for more than one array, or if you set the count, the count becomes the smallest of the possibilities.



1. Run the VI and experiment with different array sizes. Create a digital control on the front panel, wire it to the count terminal, and check the output arrays to see how different counts affect the output arrays.
2. Close the VI and do not save changes. You may not be familiar with some of the structures used in this example. They are discussed in greater detail later in this tutorial.



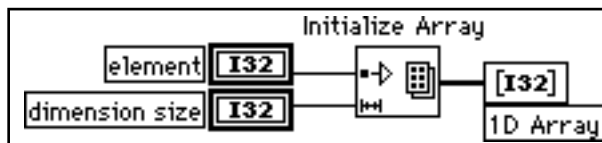
Note: *LabVIEW automatically enables Auto-indexing for every array wired to a For Loop. You can disable auto-indexing by popping up on the tunnel (entry point of the input array) and selecting **Disable Indexing**.*



LabVIEW automatically disables auto-indexing for every array wired to a While Loop. Pop up on the array tunnel of a While Loop to enable auto-indexing.

Using the Initialize Array Function

Notice that the two shift registers are initialized using the Initialize Array function, located in **Functions»Array**. Use this function to create an array whose elements all have the same value. In the following illustration, this function creates a one-dimensional array.

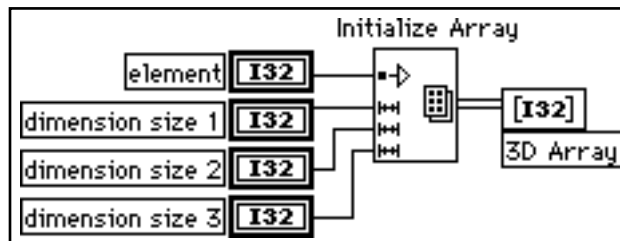


The element input determines the data type and the value of each element. The dimension size input determines the length of the array. For example, if `element` is a long integer with the value of five and `dimension size` has a value of 100, the result is a one-dimensional array of 100 long integers all set to five. You can wire the inputs from front panel control terminals, as shown in the preceding illustration, from block diagram constants, or from calculations on other parts of your diagram.



To create and initialize an array that has more than one dimension, pop up on the lower-left side of the function and select **Add Dimension**. You can also use the Resizing cursor to enlarge the Initialize Array node and add more dimension size inputs, one for each additional dimension. You can remove dimensions by shrinking the node by selecting **Remove Dimension** from the function pop-up menu or with the Resizing cursor.

The following block diagram shows how to initialize a three-dimensional array.



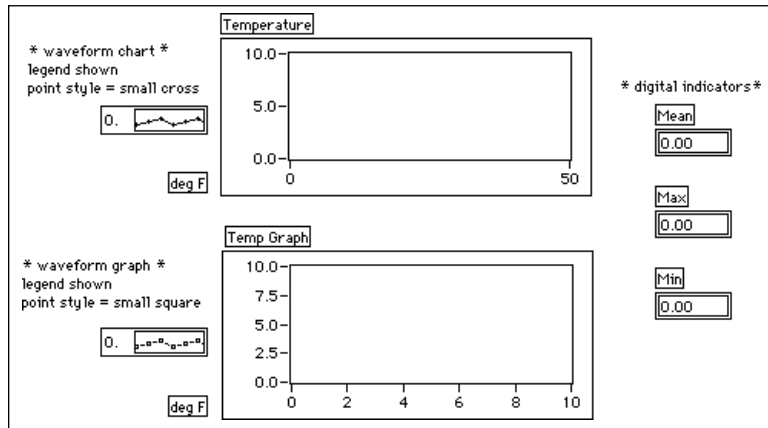
As you saw in the previous example, if all the dimension size inputs are zero, the function creates an empty array of the specified type and dimension.

Using the Graph and Analysis VIs

OBJECTIVE You will build a VI that measures temperature every 0.25 seconds for 10 seconds. During the acquisition, the VI displays the measurements in real time on a strip chart. After completing the acquisition, the VI plots the data on a graph and calculates the average, maximum, and minimum temperatures.

For examples of analysis VIs, see `examples\analysis`.

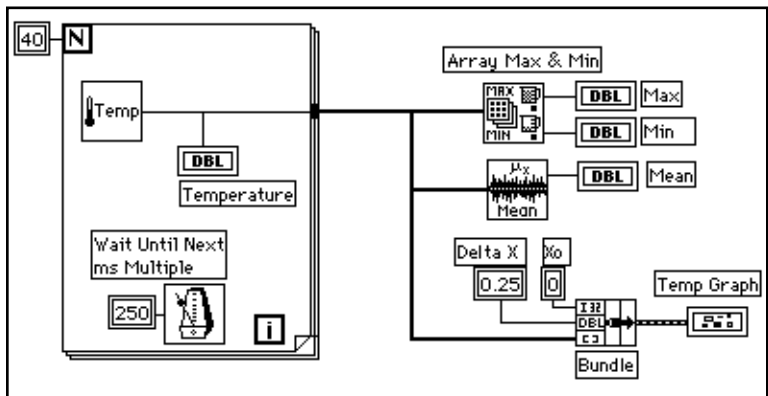
Front Panel



1. Open a new front panel and build the front panel shown in the preceding illustration. You can modify the point styles of the waveform chart and waveform graph by popping up on their legends.

The Temperature waveform chart displays the temperature as it is acquired. After acquisition, the VI plots the data in Temp Graph. The Mean, Max, and Min digital indicators display the average, maximum, and minimum temperatures.

Block Diagram



1. Build the block diagram shown in the previous illustration, using the following elements:



The Digital Thermometer VI (**Functions»Tutorial**, or you can use the VI you built in Chapter 2 by choosing **Functions»Select a VI...** and selecting My Thermometer VI. Returns one temperature measurement.



Wait Until Next ms Multiple function (**Functions»Time & Dialog**). In this exercise, this function ensures the For Loop executes every 0.25 seconds (250 milliseconds).



Numeric constant (**Functions»Numeric**). You can also pop up on the Wait Until Next ms Multiple function and select **Create Constant** to automatically create and wire the numeric constant.



Array Max & Min function (**Functions»Array**). In this exercise, this function returns the maximum and minimum temperature measured during the acquisition.



The Mean VI (**Functions»Analysis»Probability and Statistics**) returns the average of the temperature measurements.



Bundle function (**Functions»Cluster**) assembles the plot components into a cluster. The components include the initial X value (0), the delta X value (0.25), and the Y array (temperature data). Use the Positioning tool to resize the function by dragging one of the corners.

The For Loop executes 40 times. The Wait Until Next ms Multiple function causes each iteration to take place every 250 milliseconds. The VI stores the temperature measurements in an array created at the For Loop border (auto-indexing). After the For Loop completes execution, the array passes to various nodes.

The Array Max & Min function returns the maximum and minimum temperature. The Mean VI returns the average of the temperature measurements.

Your completed VI bundles the data array with an initial X value of 0 and a delta X value of 0.25. The VI requires a delta X value of 0.25 so that the VI plots the temperature array points every 0.25 seconds on the waveform graph.

2. Return to the front panel and run the VI.
3. Save the VI in mywork.llb as My Temperature Analysis.vi.

Using Arrays

LabVIEW has many functions to manipulate arrays located in **Functions»Array**. Some common functions are discussed here.

Creating and Initializing Arrays

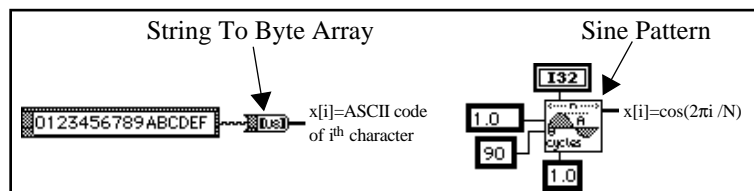
If you need an array as a source of data in your block diagram, you can choose **Functions»Array** and then select and place the array constant on your block diagram. Using the Operating tool, you can then choose a numeric constant, boolean constant, or string constant to place inside the empty array. The following illustration shows an example array constant with a numeric constant inserted into the array shell.



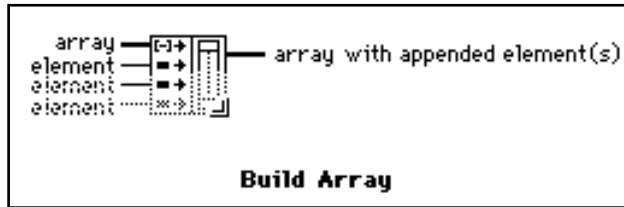
Note: *You can also create an array and its corresponding control on the front panel and then copy or drag the array control to the block diagram to create a corresponding constant.*

For information on how to create array controls and indicators on the front panel, see Chapter 15, *Array and Cluster Constants and Indicators*, in the *LabVIEW User Manual*.

There are several ways to create and initialize arrays on the block diagram. You have already seen how to create arrays at loop boundaries and how to use the Initialize Array function. Some block diagram functions also produce arrays, as the following illustration shows.



Using the Build Array Function

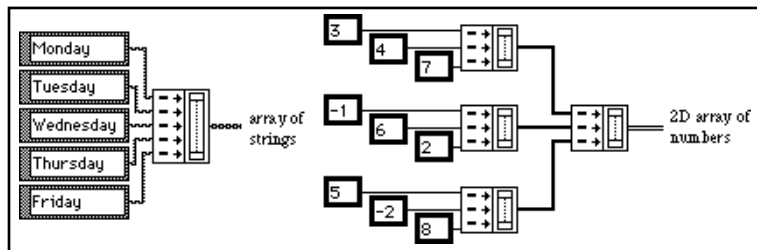


Build Array function (**Functions»Array**). You can use it to create an array from scalar values or from other arrays. Initially, the Build Array function appears with one scalar input.

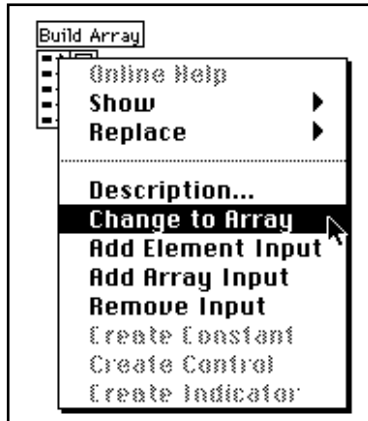


You can add as many inputs as you need to the Build Array function, and each input can be either a scalar or an array. To add more inputs, pop up on the left side of the function and select **Add Element Input** or **Add Array Input**. You can also enlarge the Build Array node with the Resizing cursor (place the Positioning tool at the corner of an object to transform it into the Resizing cursor). You can remove inputs by shrinking the node with the Resizing cursor, or by selecting **Remove Input**.

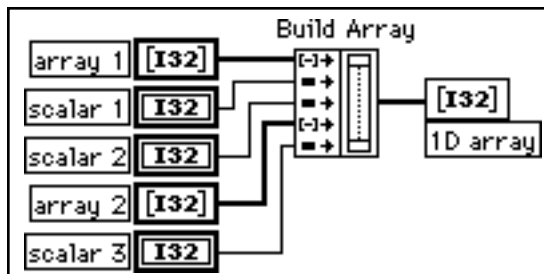
The following examples show two ways to create and initialize arrays with values from block diagram constants. On the left, five string constants are built into a one-dimensional array of strings. On the right, three groups of numeric constants are built into three, one-dimensional numeric arrays. The three arrays are then combined into a two-dimensional numeric array. The result is a 3 x 3 array with the rows 3, 4, 7; -1, 6, 2; and 5, -2, 8.



You can also create an array by combining other arrays along with scalar elements. For example, suppose you have two arrays and three scalar elements that you want to combine into a new array with the order array 1, scalar 1, scalar 2, array 2, and scalar 3. First, create a Build Array node with five inputs. Pop up on the first (top) input in the Build Array node and select **Change to Array**, as the following illustration shows. Do the same for the fourth, or next-to-last input.



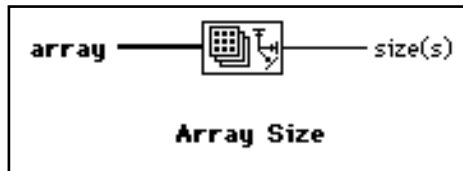
Next, wire the arrays and scalars to the node. The output array is a 1D array composed of the elements of array 1 followed by scalar 1, scalar 2, the elements of array 2, and scalar 3, as the following illustration shows.



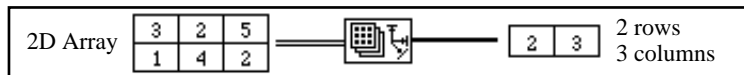
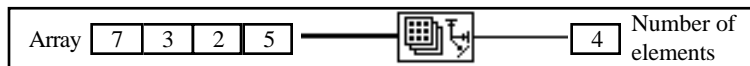
The dimension of the output array is always one dimension higher than the elements you wire to element inputs, and equal to the dimension of arrays you wire to array inputs. Element and array inputs can vary by no more than one dimension. For instance, if you wire a series of 1D arrays to element inputs, the output array consists of a 2D array

whose rows correspond to the 1D inputs. Any array inputs in this example must be 2D arrays. If element inputs are 2D arrays, then the output is a 3D array, and array inputs must be 3D arrays. You cannot build an array with scalar element inputs and 2D or higher array inputs.

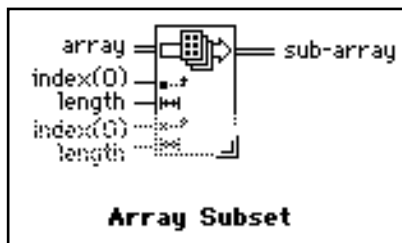
Finding the Size of an Array



Array Size returns the number of elements in the input array.

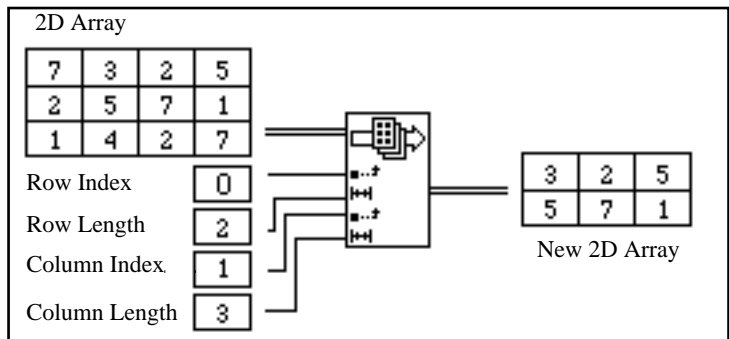
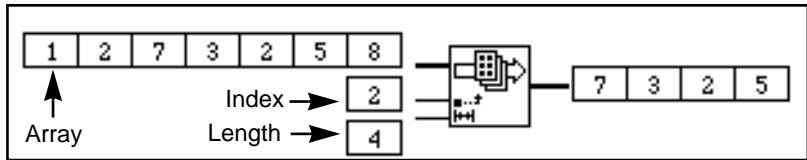


Using the Array Subset Function

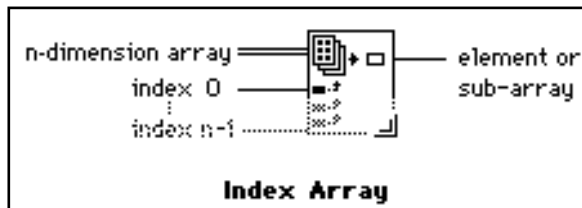


You can use this function to extract a portion of an array or matrix.

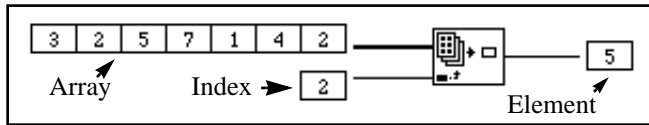
Array Subset returns a portion of an array starting at **index** and containing **length** elements. The following illustrations show examples of Array Subsets. Notice that the array index begins with 0.



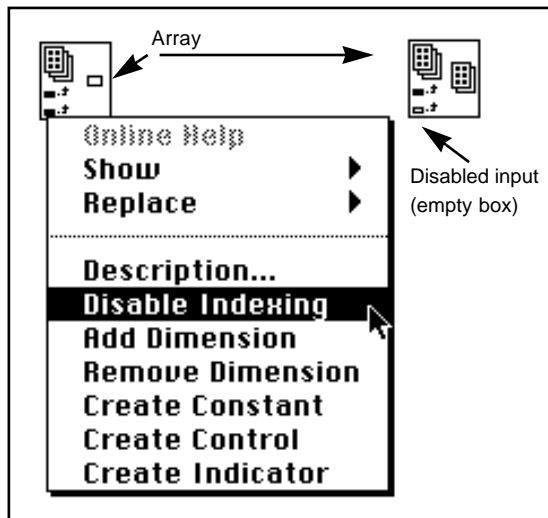
Using the Index Array Function



The Index Array function accesses an element of an array. The following illustration shows an example of an Index Array function accessing the third element of an array. Notice that the index of the third element is 2 because the first element has index 0.

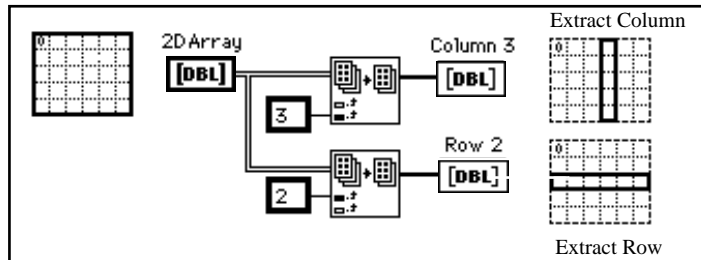


You can also use this function to *slice off* one or more dimensions of a multi-dimensional array to create a subarray of the original. To do this, stretch the Index Array function to include two index inputs, and select the **Disable Indexing** command on the pop-up menu of the second index terminal as shown in the following illustration. You have now disabled the access to a specific array column. By giving it a row index, the result is an array whose elements are the elements of the specified row of the 2D array. You can also disable indexing on the row terminal.

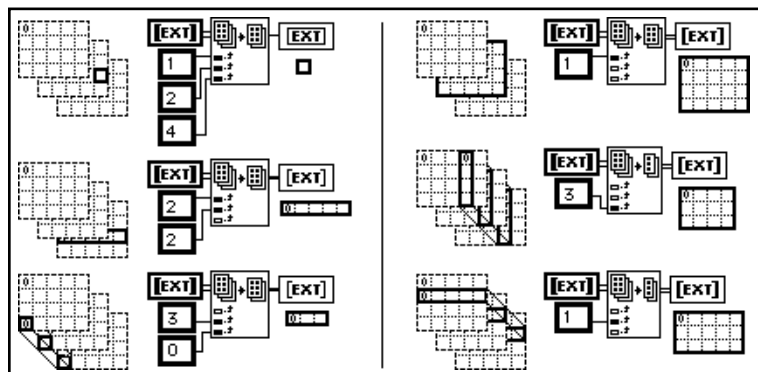


Notice that the index terminal symbol changes from a solid to an empty box when you disable indexing. To restore a disabled index, use the **Enable Indexing** command from the same menu.

You can extract subarrays along any combination of dimensions. The following example shows how to extract a one-dimensional row or column arrays from a two-dimensional array.



From a three-dimensional array, you can extract a two-dimensional array by disabling two index terminals, or a one-dimensional array by disabling a single index terminal. The following figure shows several ways to slice a three-dimensional array.



The following rules govern the use of the Index Array function to slice arrays.

1. The dimension of the output object must equal the number of disabled index terminals. For example:
 - Zero disabled—scalar element.
 - One disabled—1D component.
 - Two disabled—2D component.

2. The values wired to enabled terminals must identify the output elements.

Thus, you can interpret the lower left preceding example as a command to generate a one-dimensional array of all elements at column 0 and row 3. You can interpret the upper right example as a command to generate a two-dimensional array of page 1. The new, 0th element is the one closest to the original, as shown in the preceding illustration.

Summary

An array consists of a collection of data elements of the same type. The examples you studied in this lesson used numeric arrays. Keep in mind that arrays can be of any data type, such as Boolean or string.

You create an array on the block diagram using a two-step process. First, you place an array constant from **Functions»Array** on the block diagram, and then you add the desired constant or indicator to the array shell. Remember that you can also create an array on the front panel by selecting **Controls»Array & Cluster**, and then adding the desired control of indicator to the array shell.

Both the For Loop and the While Loop can accumulate arrays at their borders. This feature is useful when creating and processing arrays.



Note:

Remember that by default, LabVIEW enables indexing on For Loops and disables indexing on While Loops.

Polymorphism is the ability of a function to adjust to input data of different data types. All functions that accept numeric input can accept any numeric representation, an array of numerics, or a cluster of numerics.

You can plot your data using graphs. Graphs have many useful features that you can use to customize your plot display. You can display more than one plot on a graph using the Build Array function from **Functions»Array**. The graph automatically becomes a multiplot graph when you wire the array of outputs to the terminal.

Many functions manipulate arrays, such as the Build Array function and the Index Array function from **Functions»Array**. In the exercises in this chapter, you used array functions to work with only one-dimensional arrays; however, the same functions also work with multidimensional arrays.

Additional Topics

More About Arrays

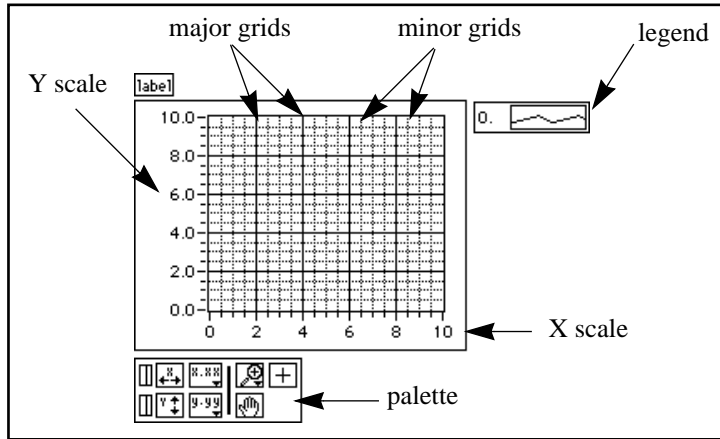
Many more array functions exist in LabVIEW than we have discussed here. These functions include Replace Array Element, Search 1D Array, Sort 1D Array, Reverse 1D Array, Multiply Array Elements, and many other array functions. For more information about arrays and the array functions available, refer to Chapter 15, *Array and Cluster Controls and Indicators*, of the *LabVIEW User Manual*; **Online Reference»Function and VI Reference»Array Functions**; and **Online Reference»Function and VI Reference»Cluster Functions**.

Efficient Memory Usage: Minimizing Data Copies

To save memory, you can use single-precision arrays instead of double-precision arrays. If you want to understand how LabVIEW uses memory, see the *Memory Usage* section in Chapter 27, *Performance Issues*, in the *LabVIEW User Manual*.

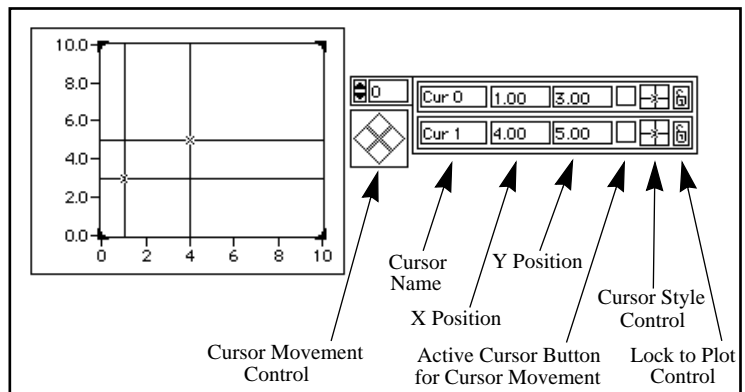
Customizing Graphs

Both waveform and XY graphs have a number of optional parts that you can show or hide using the **Show** submenu of the pop-up menu for the graph. Some of the options include a legend, through which you can define the color and style for a given plot, a palette from which you can change scaling and format options while the VI is running, and a cursor display. The following illustration of a graph shows all of the optional components except for the cursor display.



Graph Cursors

You can place cursors and a cursor display on all the graphs in LabVIEW, and you can label the cursor on the plot. LabVIEW can set and read cursors programmatically. You can set a cursor to lock onto a plot, and you can move multiple cursors at the same time. There is no limit to the number of cursors a graph can have. The following illustration shows a waveform graph with the cursor display.



For more detailed information on customizing graphs, see Chapter 16, *Graph and Chart Controls and Indicators*, in the *LabVIEW User Manual*.

Refer to the `ZoomGraph.vi` in `examples\general\graphs\zoom.llb` for an example that reads cursor values and programmatically zooms in and out of a graph using the cursors.

Intensity Plots

LabVIEW has two methods for displaying three-dimensional data: the intensity chart and the intensity graph. Both intensity plots accept two-dimensional arrays of numbers, where each number is mapped to a color. You can define the color mapping interactively, using an optional color ramp scale, or programmatically, using an attribute node for the chart. For more information about the intensity plots, see Chapter 16, *Graph and Chart Controls and Indicators*, in your *LabVIEW User Manual*. For examples using the intensity chart and graph, refer to `intgraph.llb` in the `examples\general\graphs` directory.

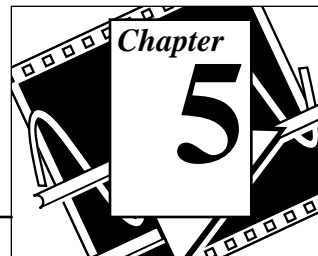
Data Acquisition Arrays (Windows, Macintosh, and Sun)

Data returned from a plug-in data acquisition board using the Data Acquisition VIs can be in the form of a single value, a one-dimensional array, or a two-dimensional array. See the *Data Organization in for Analog Applications* section in Chapter 3, *Basic Data Acquisition Concepts*, in the *LabVIEW Data Acquisition Basics Manual* for more information on manipulating data arrays.

Graph Examples

You can find a number of graph examples located in `examples\general\graphs`, which contains VIs to perform many varied functions with arrays and graphs.

Case and Sequence Structures and the Formula Node



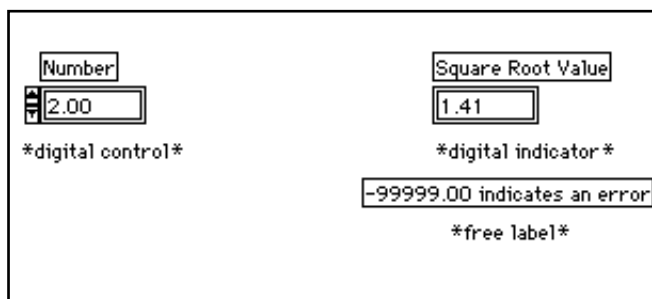
You Will Learn:

- How to use the Case structure.
- How to use the Sequence structure.
- What Sequence Locals are and how to use them.
- What a Formula Node is and how to use it.
- For examples of structures, see `examples\general\structs.llb`.

Using the Case Structure

OBJECTIVE You will build a VI that checks a number to see if it is positive. If the number is positive the VI calculates the square root of the number; otherwise, the VI returns an error.

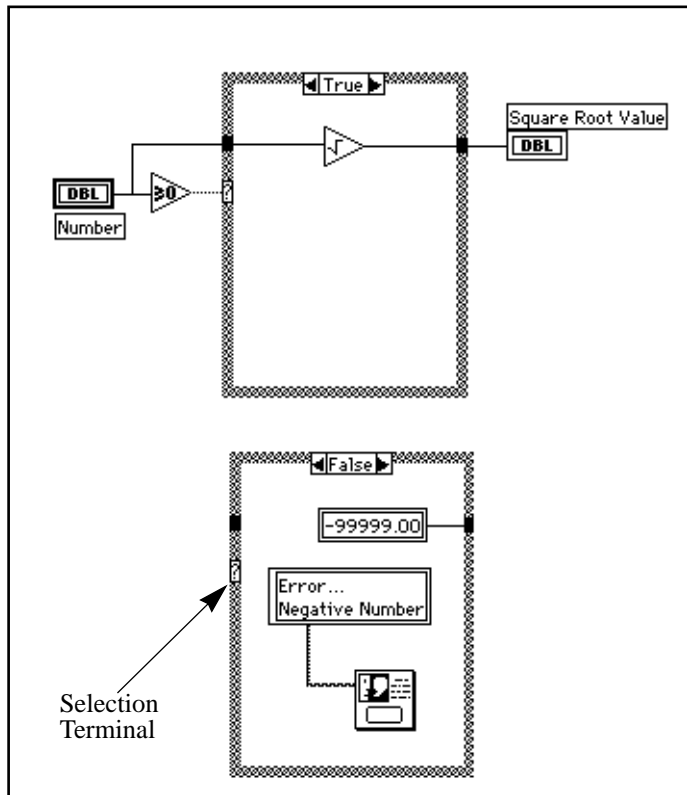
Front Panel



1. Open a new front panel and build the front panel as shown in the previous illustration.

The Number control supplies the number. The Square Root Value indicator displays the square root of the number. The free label acts as a note to the user.

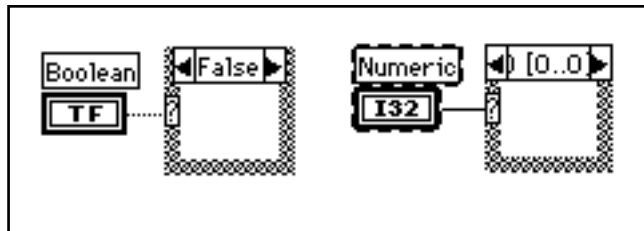
Block Diagram



1. Open the block diagram.
2. Place a Case structure (**Functions»Structures**) in the block diagram. Enlarge the Case structure by dragging one corner with the Resizing cursor.

By default, the Case structure is Boolean and it has only two cases: True and False. A Boolean Case structure is analogous to an if-then-else statement in text-based, programming languages. It

automatically changes to numeric when you wire a numeric control to the selection terminal.



You can display only one case at a time. To change cases, click on the arrows at the top of the Case structure.

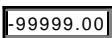
3. Select the other block diagram objects and wire them as shown in the block diagram illustration.



Greater Or Equal To 0? function (**Functions»Comparison**). In this exercise, the function determines whether the number input is negative. The function returns a TRUE if the number input is greater than or equal to 0.



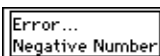
Square Root function (**Functions»Numeric**). In this exercise, the function returns the square root of the input number.



Numeric Constant (**Functions»Numeric**).



One Button Dialog function (**Functions»Time & Dialog**). In this exercise, the function displays a dialog box that contains the message Error...Negative Number.



String Constant (**Functions»String**). Enter text inside the box with the Labeling tool.

In this exercise, the VI executes either the True case or the False case. If the number is greater than or equal to zero, the VI executes the True case and returns the square root of the number. The False case outputs -99999.00 and displays a dialog box with the message Error...Negative Number.

**Note:**

You must define the output tunnel for each case. When you create an output tunnel in one case, tunnels appear at the same position in all the other cases. Unwired tunnels appear as white squares.

Be sure to wire to the output tunnel for each unwired case, clicking on the tunnel itself each time. In this exercise, you assign a value to the output tunnel in the False case because the True case has an output tunnel. If you do not want to assign the output in all cases to a value, then you must put the indicator in that case or use a global or local variable.

4. Return to the front panel and run the VI. Try a number greater than zero and a number less than zero by changing the value in the digital control you labeled Number. Notice that when you change the digital control to a negative number, LabVIEW displays the error message you set up in the False case of the case structure.
5. Save and close the VI. Name it My Square Root.vi.

VI Logic

```

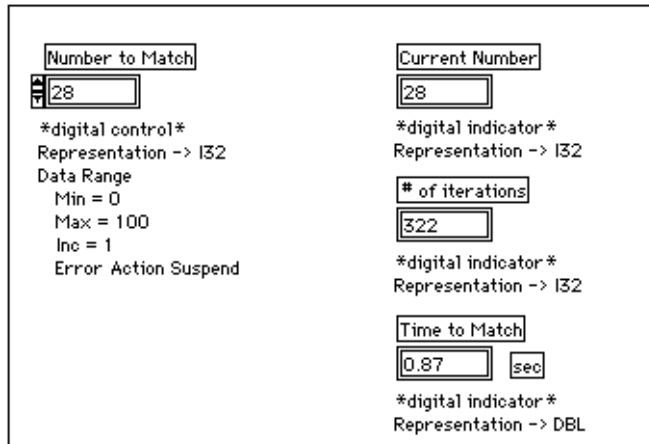
if (Number >= 0) then
Square Root Value = Sqrt(Number)
else
Square Root Value = -99999.00
Display Message "Error...Negative Number"
end if

```

Using the Sequence Structure

OBJECTIVE You will build a VI that computes the time it takes to generate a random number that matches a given number.

Front Panel



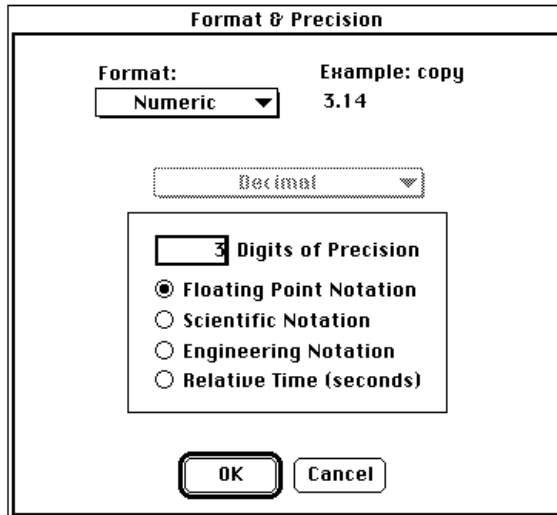
1. Open a new front panel and build the front panel shown in the following illustration. Be sure to modify the controls and indicators as described in the text following the illustration.

The **Number to Match** control contains the number you want to match. The **Current Number** indicator displays the current random number. The **# of iterations** indicator displays the number of iterations before a match. **Time to Match** indicates how many seconds it took to find the matching number.

Modifying the Numeric Format

By default, LabVIEW displays values in numeric controls in decimal notation with two decimal places (for example, 3.14). You can use the **Format & Precision...** option of a control or indicator pop-up menu to change the precision or to display the numeric controls and indicators in scientific or engineering notation. You can also use the **Format & Precision...** option to denote time and date formats for numerics.

1. Change the precision on the Time to Match indicator.
 - a. Pop up on the Time to Match digital indicator and choose **Format & Precision...**. You must be in the front panel to access the menu.
 - b. Enter a 3 for Digits of Precision and click on **OK**.



2. Change the representation of the digital control and two of the digital indicators to long integers.
 - a. Pop up on the Number to Match digital control and choose **Representation»Long**.
 - b. Repeat the previous step for the Current Number, and the # of iterations digital indicators.


Setting the Data Range

With the **Data Range...** option you can prevent a user from setting a control or indicator value outside a preset range or increment. Your options are to ignore the value, coerce it to within range, or suspend execution. The range error symbol appears in place of the run button, in the toolbar, when a range error suspends execution. Also, a solid, dark border frames the control that is out of range.



1. Set the data range between 0 and 100 with an increment of 1.
 - a. Pop up on the Time to Match indicator and choose **Data Range...**
 - b. Fill in the dialog box, as shown in the following illustration, and click on **OK**.

Representation



Long

Minimum

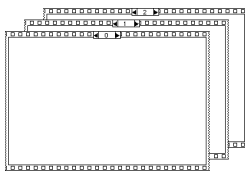
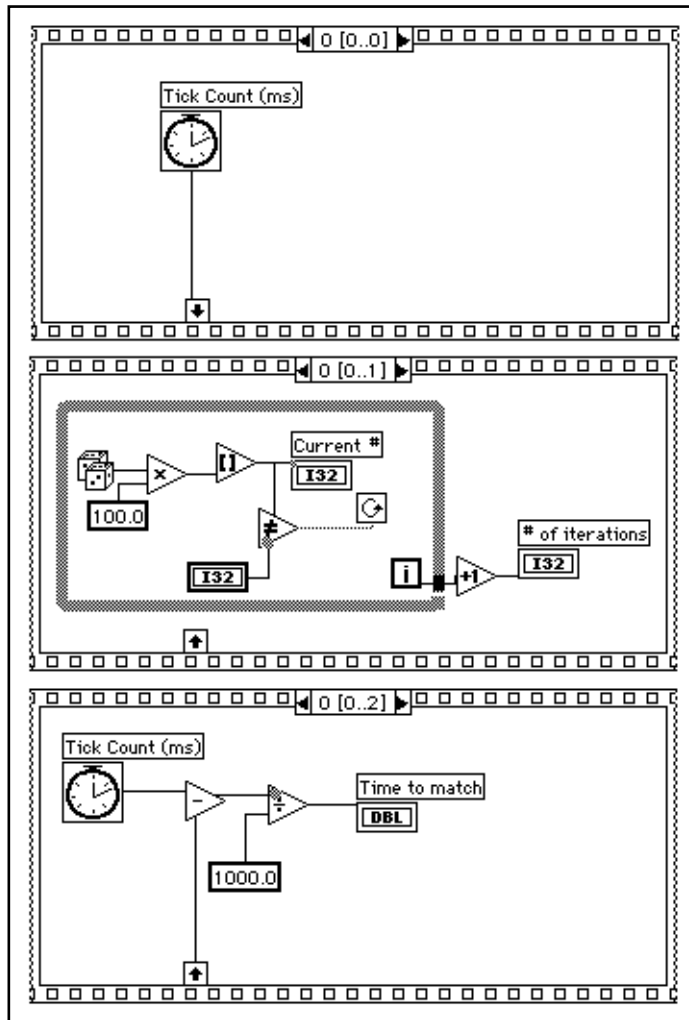
Maximum

Increment

Default

If Value is Out of Range:

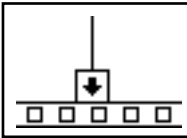
Block Diagram



1. Open the block diagram.
2. Place the Sequence structure (**Functions»Structures**) in the block diagram.

The Sequence structure, which looks like frames of film, executes block diagrams sequentially. In conventional programming languages, the program statements execute in the order in which they appear. In data flow programming, a node executes when data is available at all of the node inputs, although sometimes it is necessary to execute one node before another. LabVIEW uses the Sequence structure as a method to control the order in which nodes execute. LabVIEW places the diagram that the VI executes first inside the border of Frame 0, it places the diagram it executes second inside the border of Frame 1, and so on. As with the Case structure, only one frame is visible at a time.

3. Enlarge the structure by dragging one corner with the Resizing cursor.
4. Create a new frame by popping up on the frame border and choose **Add Frame After**. Repeat this step to create Frame 2.



Frame 0 in the previous illustration contains a small box with an arrow in it. That box is a sequence local variable which passes data between frames of a Sequence structure. You can create sequence locals on the border of a frame. The data wired to a frame sequence local is then available in subsequent frames. However, you cannot access the data in frames preceding the frame in which you created the sequence local.

5. Create the sequence local by popping up on the bottom border of Frame 0 and choosing **Add Sequence Local**.

The sequence local appears as an empty square. The arrow inside the square appears automatically when you wire a function to the sequence local.

6. Finish the block diagram as shown in the opening illustration of the *Block Diagram* section.



Tick Count (ms) function (**Functions»Time & Dialog**). Returns the number of milliseconds that have elapsed since power on. For this exercise, you need two Tick Count functions.



Random Number (0-1) function (**Functions»Numeric**). Returns a random number between 0 and 1.



Multiply function (**Functions»Numeric**). In this exercise, the function multiplies the random number by 100. In other words, the function returns a random number between 0.0 and 100.0.



Numeric Constant function (**Functions»Numeric**). In this exercise, the numeric constant represents the maximum number that can be multiplied.



Round to Nearest function (**Functions»Numeric**). In this exercise, the function rounds the random number between 0 and 100 to the nearest whole number.



Not Equal? function (**Functions»Comparison**). In this exercise, the function compares the random number to the number specified in the front panel and returns a TRUE if the numbers are not equal. Otherwise, this function returns FALSE.



Increment function (**Functions»Numeric**). In this exercise, the function increments the While Loop count by 1.



Subtract function (**Functions»Numeric**). In this exercise, the function returns the time (in milliseconds) elapsed between Frame 2 and Frame 0.



Divide function (**Functions»Numeric**). In this exercise, the function divides the number of milliseconds elapsed by 1000 to convert the number to seconds.



Numeric constant (**Functions»Numeric**). In this exercise, the function converts the number from milliseconds to seconds.

In Frame 0, the Tick Count (ms) function returns the current time in milliseconds. This value is wired to the sequence local, where the value is available in subsequent frames. In Frame 1, the VI executes the While Loop as long as the number specified does not match the number that the Random Number (0-1) function returns. In Frame 2, the Tick Count (ms) function returns a new time in milliseconds. The VI subtracts the old time (passed from Frame 0 through the Sequence local) from the new time to compute the time elapsed.

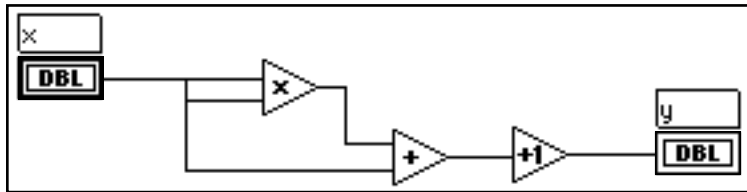
7. Return to the front panel and enter a number inside the Number to Match control and run the VI.
8. Save and close the VI. Name it `My Time to Match.vi`.

Formula Node

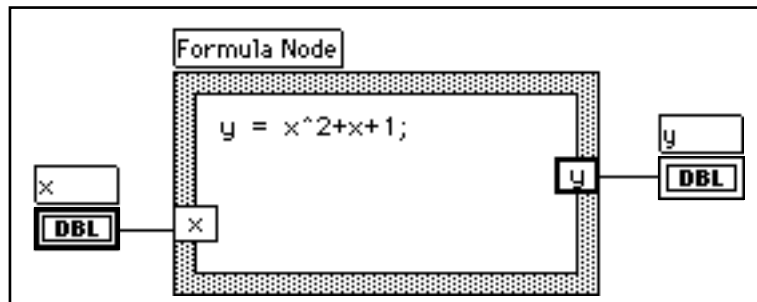
The *Formula Node* is a resizable box that you can use to enter formulas directly into a block diagram. You place the Formula Node on the block diagram by selecting it from **Function»Structures**. This feature is useful when an equation has many variables or is otherwise complicated. For example, consider the equation:

$$y = x^2 + x + 1.$$

If you implement this equation using regular LabVIEW arithmetic functions, the block diagram looks like the one in the following illustration.



You can implement the same equation using a Formula Node, as shown in the following illustration.



With the Formula Node, you can directly enter a complicated formula, or formulas, in lieu of creating block diagram subsections. You enter formulas with the Labeling tool. You create the input and output terminals of the Formula Node by popping up on the border of the node and choosing **Add Input (Add Output)**. Type the variable name in the box. Variables are case sensitive. You enter the formula or formulas inside the box. Each formula statement must end with a semicolon (;).

The operators and functions available inside the Formula Node are listed in the Help window for the Formula Node, as shown in the following illustration. A semicolon terminates each formula statement. Formula Node functions are described in more detail in Chapter 20, *The Formula Node*, of the *LabVIEW User Manual*.

```

Formula Node operators, lowest precedence first:
=                assignment
? :             conditional
|| &&           logical
== != > < >= <= relational
+ - * / ^       arithmetic
+ - !           unary

Formula Node functions:
abs acos acosh asin asinh atan atanh ceil
cos cosh cot csc exp expm1 floor getexp getman
int intrz ln ln1 log log2 max min mod rand
rem sec sign sin sinc sinh sqrt tan tanh

```

The following example shows how you can perform a conditional assignment inside a Formula Node.

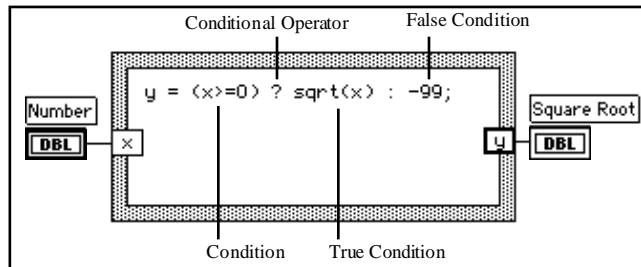
Consider the following code fragment that computes the square root of x if x is positive, and assigns the result to y . If x is negative, the code assigns -99 to y .

```

if (x >= 0) then
y = sqrt(x)
else
y = -99
end if

```

You can implement the code fragment using a Formula Node, as shown in the following diagram.



Using the Formula Node

OBJECTIVE You will build a VI that uses the Formula Node to calculate the following equations.

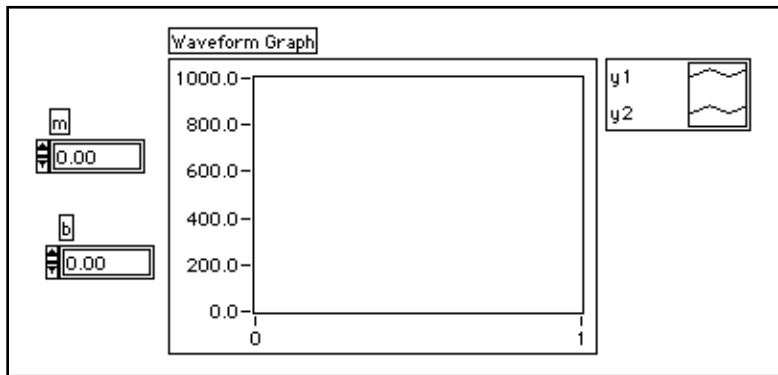
$$y1 = x^3 - x^2 + 5$$

$$y2 = m * x + b$$

where x ranges from 0 to 10.

You will use only one Formula Node for both equations, and you will graph the results on the same graph.

Front Panel

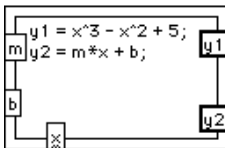
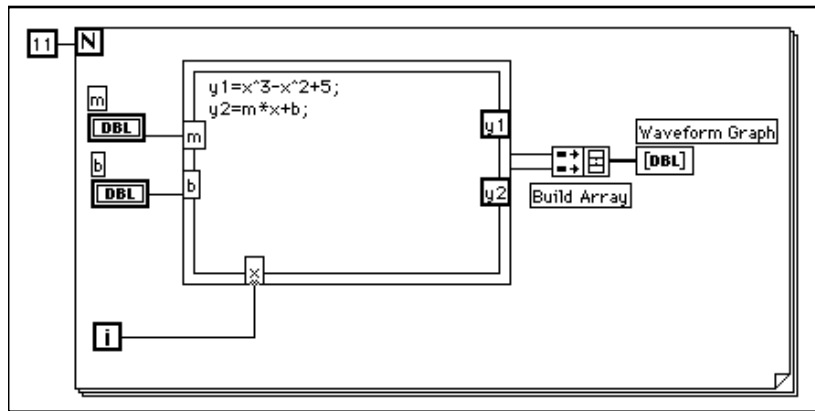


1. Open a new front panel and build the front panel shown in the preceding illustration. The waveform graph indicator displays the plots of the equation. The VI uses the two digital controls to input the values for m and b .



Create the graph legend shown in the preceding illustration by selecting **Show»Legend**. Use the Resizing cursor to drag the legend downward so it displays two plots. Use the Labeling tool to rename the plots. You can define the line style for each plot using the legend pop-up menu. You can also color each plot by popping up on the legend and choosing **Color**.

Block Diagram



1. Build the block diagram shown in the preceding illustration.
2. Place the For Loop (**Functions»Structures**) in the block diagram and drag the corner to enlarge the loop.

Formula Node (**Functions»Structures**). With this node, you can directly enter formula(s). Create the three input terminals by popping up on the border and choosing **Add Input**. You create the output terminal by choosing **Add Output** from the pop-up menu.

When you create an input or output terminal, you must give it a variable name. The variable name must exactly match the one you use in the formula. The names are case sensitive. That is, if you use a lower case a in naming the terminal, you must use a lower case a in the formula. You can enter the variable names and formula with the Labeling tool.



Note:

Although variable names are not limited in length, be aware that long names take up considerable diagram space. A semicolon (;) terminates the formula statement.



Numeric Constant (**Functions»Numeric**). You can also pop up on the count terminal and select **Create Constant** to automatically create and wire the numeric constant. The numeric constant specifies the number of For Loop iterations. If x range is 0 to 10 including 10, you need to wire 11 to the count terminal.



Because the iteration terminal counts from 0 to 10, you use it to control the X value in the Formula Node.



Build Array (**Functions»Array**) puts two array inputs into the form of a multiplot graph. Create the two input terminals by using the Resizing cursor to drag one of the corners.

3. Return to the front panel and run the VI with different values for m and b .
4. Save and close the VI. Name the VI `My Equations.vi`.

Summary

Two of the LabVIEW structures that control data flow are the Case structure and the Sequence structure. LabVIEW depicts both structures like a deck of cards; only one case or one frame is visible at a time.

You use the Case structure to branch to different subdiagrams depending on the input to the selection terminal of the Case structure. You place the subdiagrams inside the border of each case of the Case structure. The case selection can be either Boolean (2 cases) or numeric (up to $2^{31} - 1$ cases). LabVIEW automatically determines the selection terminal type when you wire a Boolean or numeric control to it.

You use the Sequence structure to execute the diagram in a specific order. You place the portion of the diagram that you want to execute first in frame 0 of the Sequence structure, the diagram that you want to execute second in frame 1, and so on.

You use sequence locals to pass values between Sequence structure frames. The data passed in a sequence local is available only in frames subsequent to the frame in which you created the sequence local, and not in those frames that precede the frame.

With the Formula Node, you can directly enter formulas in the block diagram. This feature is useful when a function equation has many variables or is complicated. Remember that variable names are case sensitive and that each formula statement must end with a semicolon (;).

Additional Topics

More Information on Case and Sequence Structures

For more information on Case and Sequence structures, see the *Case and Sequence Structures* section, in Chapter 19, *Structures*, of the *LabVIEW User Manual*. Also, because these structures are fundamental LabVIEW programming elements, you can see their use in many of the VIs in the `examples` directory.

Timing with Sequence Structures

One common use of Sequence structures is to calculate the execution time of a function or VI. The `Timing Template.vi` example in `examples\general\structs.llb` includes a template for this operation.

More Information on Formula Nodes

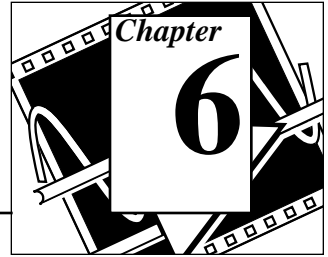
You can find more information on Formula Nodes in Chapter 20, *The Formula Node*, of the *LabVIEW User Manual*.

Artificial Data Dependency

Nodes not connected by a wire can execute in any order. Nodes do not necessarily execute in left-to-right, top-to-bottom order. A Sequence structure is one way to control execution order when natural data dependency does not exist.

Another way to control execution order is to create an *artificial data dependency*, a condition in which the *arrival* of data rather than its value triggers execution of an object. The receiver may not actually use the data internally. The advantage of artificial dependency is that all of the nodes are visible at one level, although, in some cases, the confusion created by the artificial links between nodes can be a disadvantage.

You can open the `Timing Template (data dep).vi` from `examples\general\structs.llb` to see how the `Timing Template` has been altered to use artificial data dependency rather than a sequence structure.



Strings and File I/O

You Will Learn:

- How to create string controls and indicators.
- How to use string functions.
- About file input and output operations.
- How to save data to files in spreadsheet format.
- How to write data to and read data from text files.

Strings

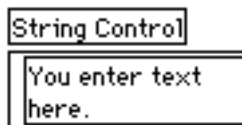
A string is a collection of ASCII characters. You can use strings for more than simple text messages. In instrument control, you can pass numeric data as character strings and then convert these strings to numbers. Storing numeric data to disk can also involve strings. To store numbers in an ASCII file, you must first convert numbers to strings before writing the numbers to a disk file.

For examples of strings, see `examples\general\strings.llb`.

Creating String Controls and Indicators

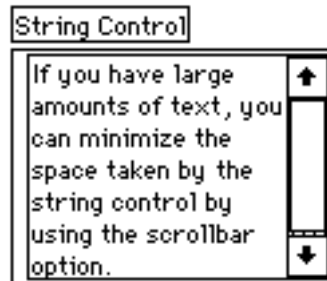


You can find the string control and indicator, shown at left, in **Controls»String & Table**. You can enter or change text inside a string control using the Operating tool or the Labeling tool. Enlarge string controls and indicators by dragging a corner with the Positioning tool.



Strings and File I/O

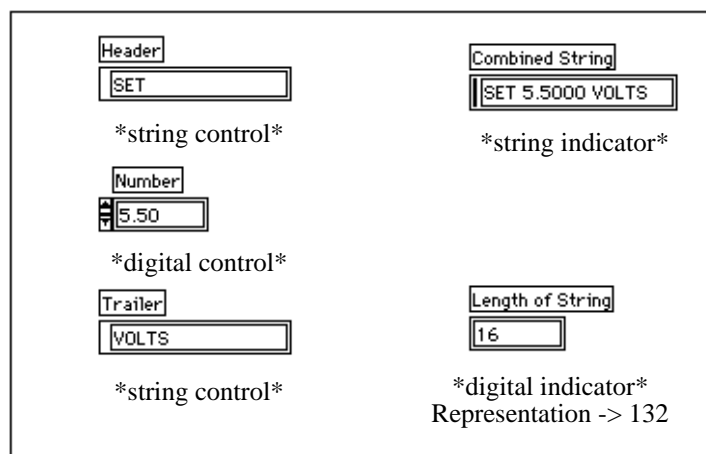
If you want to minimize space that a front panel string control or indicator occupies, select **Show»Scrollbar**. If this option is dimmed, you must increase the vertical size of the window to make it available.



Using String Functions

OBJECTIVE LabVIEW has many functions to manipulate strings. You will find these functions in **Functions»String**. You will build a VI that converts a number to a string and concatenates the string with other strings to form a single output string. The VI also determines the output string length.

Front Panel

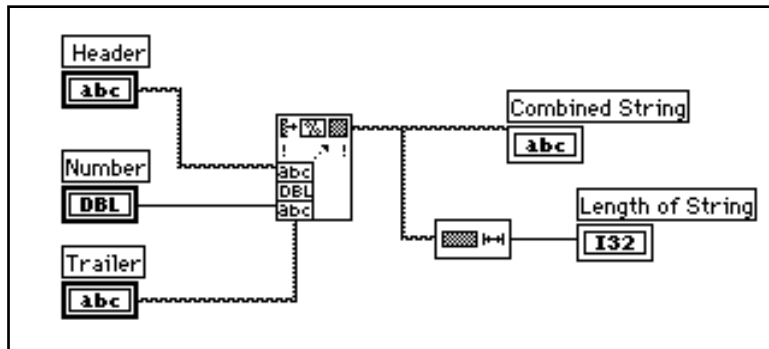


1. Open a new front panel and build the front panel shown in the preceding illustration. Be sure to modify the controls and indicators as depicted.

The two string controls and the digital control can be combined into a single output string and displayed in the string indicator. The digital indicator displays the string length.

The Combined String output in this exercise has a similar format to command strings used to communicate with GPIB (IEEE 488) and serial (RS-232 or RS-422) instruments. Refer to Chapter 8, *Data Acquisition and Instrument Control*, of this tutorial to learn more about strings used for instrument commands.

Block Diagram



1. Build the block diagram shown in the preceding illustration.



Format Into String function (**Functions»String**) concatenates and formats numbers and strings into a single output string. Use the Resizing cursor on the icon to add three argument inputs.



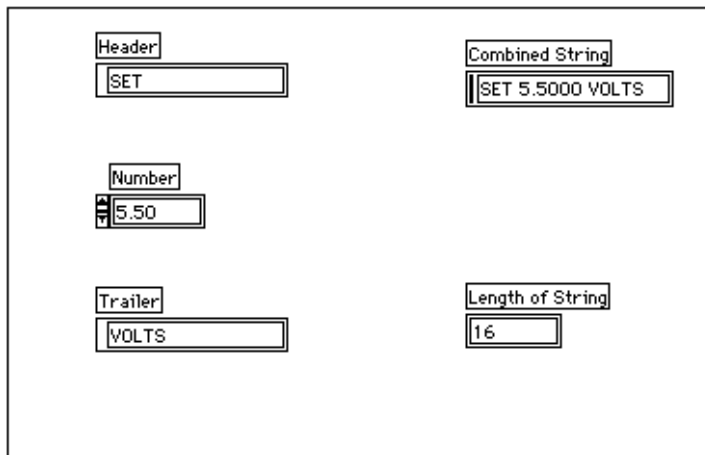
String Length function (**Functions»String**) returns the number of characters in the concatenated string.

2. Run the VI. Notice that the Format Into String function concatenates the two string controls and the digital control into a single, output string.
3. Save the VI as `My Build String.vi`. You will use this VI in the next exercise.

Using Format Strings

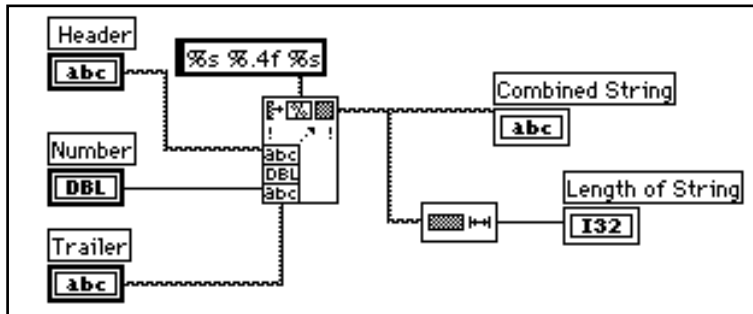
OBJECTIVE You will use the My Build String VI that you created in the previous exercise to create a format string. With format strings, you can specify the format of arguments, including the field width, base (hex, octal, and so on), and any text that separates the arguments.

Front Panel

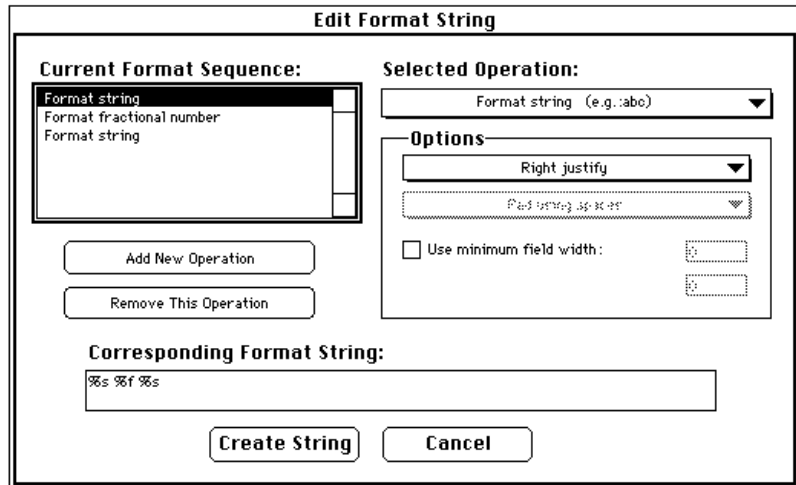


1. Open the My Build String VI that you created in the previous exercise.

Block Diagram



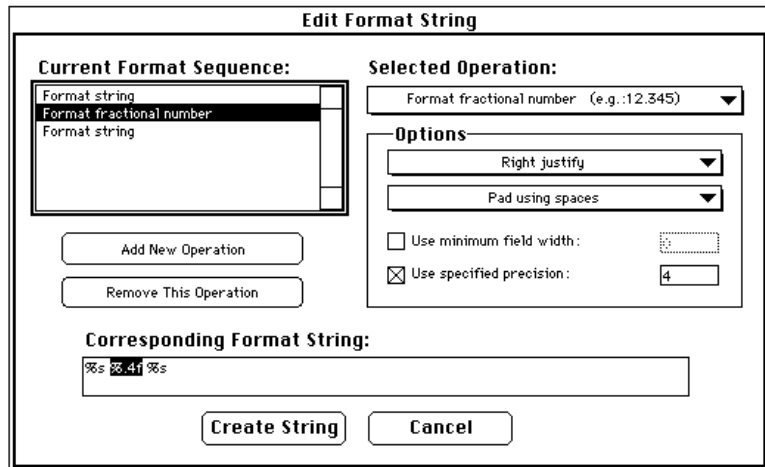
1. Pop up on Format Into String and select **Edit Format String**. The following dialog box appears.



Note: *You can also double-click on the node to access the Edit Format String dialog box.*

Notice that the Current Format Sequence contains the argument types, in the order that you wired them.

2. Set the precision of the numeric to 4.
 - a. Highlight `Format fractional number` in the `Current Format Sequence` list box.
 - b. Click in the `Use Specified Precision` checkbox.
 - c. Highlight the numeric beside the `Use Specified Precision` checkbox, type in 4, and press `<Enter>` (Windows); `<return>` (Macintosh); `<Return>` (Sun); or `<Enter>` (HP-UX). The following illustration shows the selected options to set the precision of number.

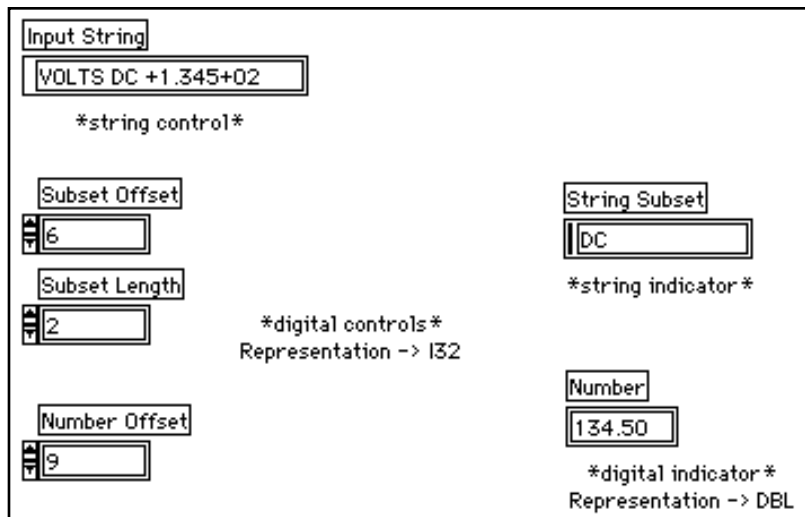


3. Press the **Create String** button. Pressing this button automatically inserts the correct format string information and wires format string to the function.
4. Return to the front panel and type text inside the two string controls and a number inside the digital control. Run the VI.
5. Save and close the VI. Name it `My Format String.vi`.

More String Functions

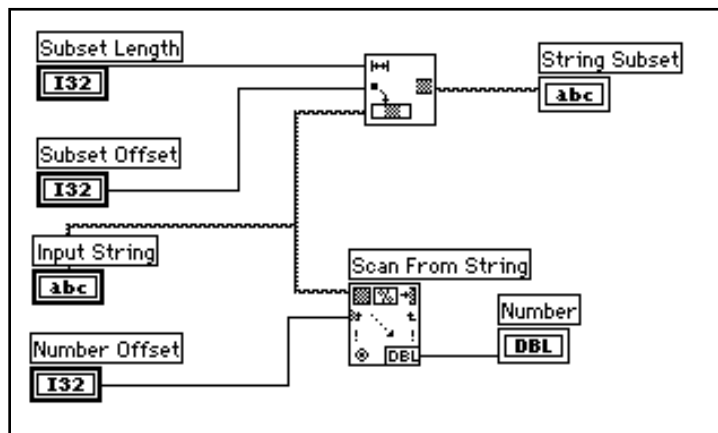
OBJECTIVE In the previous exercise, you used the string functions to create a long string from numbers and smaller strings. In the following exercise, you will examine a VI that parses information out of a longer string. You will take a subset of a string that contains the string representation of a number and convert it to a numeric value.

Front Panel



1. Open the Parse String.vi from examples\general\strings.llb. Run the VI with the default inputs. Notice that the string subset of *DC* is chosen for the input string. Also, notice that the numeric part of the string was parsed out and converted to a number. You can try different control values (remember that strings, like arrays, are indexed starting with zero), or you can show the block diagram to see how to parse the components out of the input string.

Block Diagram



1. Open the block diagram of the Parse String VI, shown in the preceding illustration.

LabVIEW uses the String Subset and From Exponential/Fract/Eng functions to parse the input string.



String Subset function (**Functions»String**) returns the substring beginning at **offset** and contains **length** number of characters. The first character offset is zero.

In many instances, you must convert strings to numbers, such as when you convert a data string received from an instrument into the data values.



Scan From String function (**Functions»String**) scans a string and converts valid, numeric characters (0 to 9, +, -, e, E, and period) to numbers. If you wire a format string, Scan From String makes conversions according to the format. If you do not wire format string, Scan From String makes default conversions for each default input terminal in the function. This function starts scanning the **string** at **offset**. The first character offset is zero.

The Scan From String function is useful when you know the header length (VOLTS DC in the example here), or when the string contains only valid numeric characters.

2. Close the VI by selecting **File»Close**. Do not save the VI.

File I/O

The LabVIEW file I/O functions (**Functions»File I/O**) are a powerful and flexible set of tools for working with files. In addition to reading and writing data, the LabVIEW file I/O functions move and rename files and directories, create spreadsheet-type files of readable ASCII text, and write data in binary form for speed and compactness.

You can store or retrieve data from files in three different formats.

- **ASCII Byte Stream.** You should store data in ASCII format when you want to access it from another software package, such as a word processing or spreadsheet program. To store data in this manner, you must convert all data to ASCII strings.
- **Datalog files.** These files are in binary format that only LabVIEW can access. Datalog files are similar to database files because you can store several different data types into one (log) record of a file.
- **Binary Byte Stream.** These files are the most compact and fastest method of storing data. You must convert the data to binary string format and you must know exactly what data types you are using to save and retrieve the data to and from files.

This section discusses ASCII byte stream files because that is the most common data file format. See the *Additional Topics* section at the end of this chapter for more information on the other two types of files.

For examples of file I/O, see `examples\file`.

File I/O Functions

Most file I/O operations involve three basic steps: opening an existing file or creating a new file; writing to or reading from the file; and closing the file. Therefore, LabVIEW contains many utility VIs in **Function»File I/O**. This section describes the nine, high-level utilities. These utility functions are built upon intermediate-level VIs that incorporate error checking and handling with the file I/O functions.

You can also set a delimiter or string of delimiters, such as tabs, commas, and so on, in your spreadsheet. This saves you from parsing your spreadsheet if you used a delimiter other than the default tab to set up the spreadsheet.



The Write Characters To File VI writes a character string to a new byte stream file or appends the string to an existing file. This VI opens or creates the file, writes the data, and then closes the file.



The Read Characters From File VI reads a specified number of characters from a byte stream file beginning at a specified character offset. This VI opens the file beforehand and closes it afterwards.



The Read Lines From File VI reads a specified number of lines from a byte stream file beginning at a specified character offset. This VI opens the file beforehand and closes it afterwards.



The Write To Spreadsheet File VI converts a 1D or 2D array of single-precision numbers to a text string and writes the string to a new byte stream file or appends the string to an existing file. You can optionally transpose the data. This VI opens or creates the file beforehand and closes it afterwards. You can use this VI to create text files readable by most spreadsheet programs.



The Read From Spreadsheet File VI reads a specified number of lines or rows from a numeric text file, beginning at a specified character offset, and converts the data to a 2D, single-precision array of numbers. You can optionally transpose the array. This VI opens the file beforehand and closes it afterwards. You can use this VI to read spreadsheet files saved in text format.

For additional File I/O functions, select **Function»File I/O»Binary File VIs** or **Function»File I/O»Advanced File Functions**.

Writing to a Spreadsheet File

One very common application for saving data to a file is to format the text file so that you can open it in a spreadsheet. In most spreadsheets, tabs separate columns and EOL (End of Line) characters separate rows, as shown in the following figure.

```

0.00↵0.4258↵  ↵ = Tab
1.00↵0.3073↵  ↵ = Line Separator
2.00↵0.9453↵
3.00↵0.9640↵
4.00↵0.9517↵

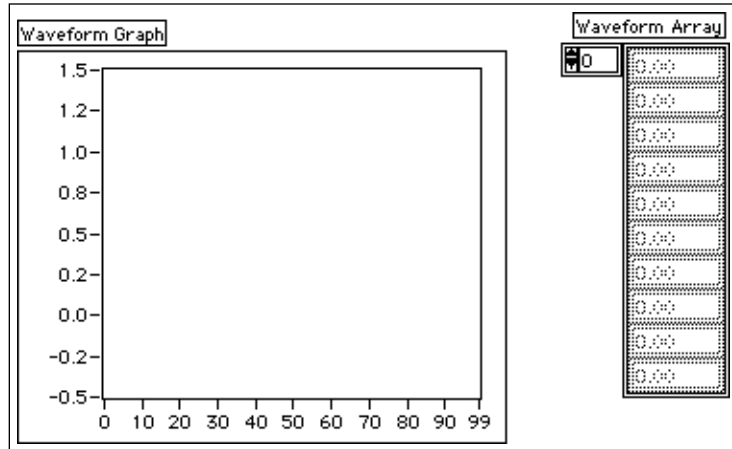
```

Opening the file using a spreadsheet program yields the following table.

	A	B	C
1	0	0.4258	
2	1	0.3073	
3	2	0.9453	
4	3	0.964	
5	4	0.9517	
6			

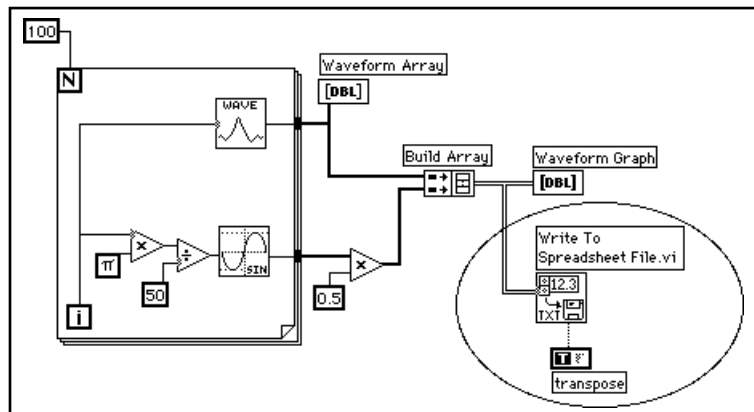
OBJECTIVE You will modify an existing VI to use a file I/O function so that you can save data to a new file in ASCII format. Later you can access this file from a spreadsheet application.

Front Panel



1. Open the `My Graph Waveform Arrays.vi` you built in Chapter 4 of this tutorial. As you recall, this VI generates two data arrays and plots them on a graph. You modify this VI to write the two arrays to a file where each column contains a data array.

Block Diagram



2. Open the block diagram of `My Graph Waveform Arrays` and modify the VI by adding the block diagram functions that have been placed inside the oval, as shown in the preceding illustration.



The Write To Spreadsheet File VI (**Functions»File I/O**) converts the two-dimensional array to a spreadsheet string and writes it to a file. If you have not specified a path name, then a file dialog box pops up and prompts you for a file name. The Write To Spreadsheet File writes either a 1-dimensional or 2-dimensional array to file. Because you have a 2D array of data in this example, you do not have to wire to the 1D input. With this VI, you can use a spreadsheet delimiter or string of delimiters, such as tabs or commas in your data.



Boolean Constant (**Functions»Boolean**) controls whether or not LabVIEW transposes the 2D array before writing it to file. To change the value to TRUE click on the constant with the Operating tool. In this case, you want the data transposed because the data arrays are row specific (each row of the two-dimensional array is a data array). Because each column of the spreadsheet file contains a data array, the 2D array must first be transposed.

- Return to the front panel and run the VI. After the data arrays have been generated, a file dialog box prompts you for the file name of the new file you are creating. Type in a file name and click on **OK**.



Caution: *Do not attempt to write data in VI libraries, such as the mywork.llb. Doing so may result in overwriting your library and losing your previous work.*

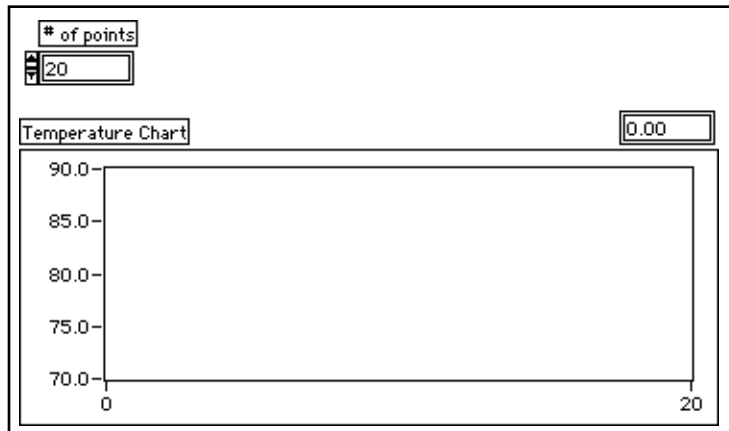
- Save the VI, name it My Waveform Arrays to File.vi, and close the VI.
- You now can use spreadsheet software or a text editor to open and view the file you just created. You should see two columns of 100 elements.

In this example, the data was not converted or written to file until the entire data arrays had been collected. If you are acquiring large buffers of data or would like to write the data values to disk as they are being generated, then you must use a different File I/O VI.

Appending Data to a File

OBJECTIVE You will create a VI to append temperature data to a file in ASCII format. This VI uses a For Loop to generate temperature values and store them in a file. During each iteration, you will convert the data to a string, add a comma as a delimiting character, and append the string to a file.

Front Panel



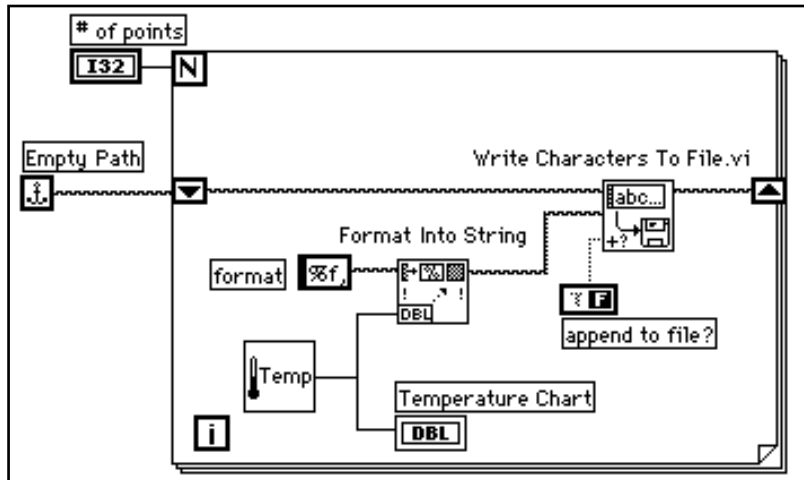
1. Open a new front panel and place the objects as shown in the preceding illustration.

The front panel contains a digital control and a waveform chart. Select **Show»Digital Display**. The # of points control specifies how many temperature values to acquire and write to file. The chart displays the temperature curve. Rescale the y axis of the chart for the range 70.0 to 90.0, and rescale the x axis for the range 0 to 20.



2. Pop up on the # of points digital control and choose **Representation»Long**.

Block Diagram



1. Open the block diagram.
2. Add the For Loop and enlarge it. This VI generates the number of temperature values specified by the # of Points control.
3. Add a Shift Register to the loop by popping up on the loop border. This shift register contains the path name to the file.
4. Finish wiring the objects.



Empty Path constant (**Functions»File I/O»File Constants**). The Empty Path function initializes the shift register so that the first time you try to write a value to file, the path is empty. A file dialog box prompts you to enter a file name.



The My Thermometer VI you built in Chapter 2 (**Functions»Select a VI...**) or the Digital Thermometer VI (**Functions»Tutorial**) returns a simulated temperature measurement from a temperature sensor.



Format Into String function (**Functions»String**) converts the temperature measurement (a number) to a string and concatenates the comma that follows it.



String constant (**Functions»String**). This format string specifies that you want to convert a number to a fractional format string and follow the string with a comma.



The Write Characters To File VI (**Functions»File I/O**) writes a string of characters to a file.



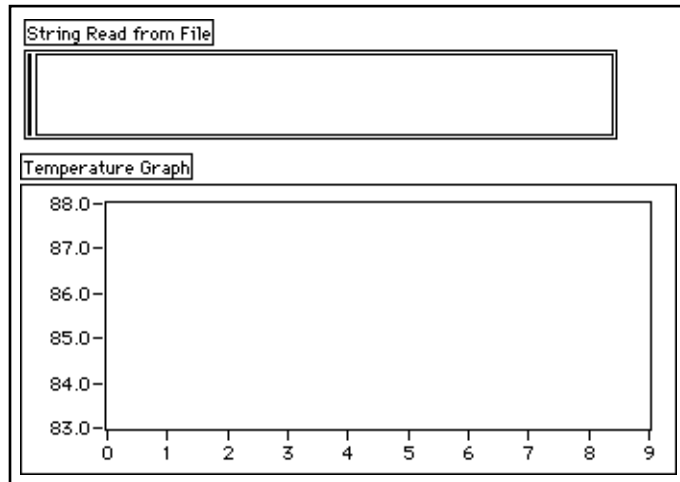
Boolean Constant (**Functions»Boolean**) sets the `append to file?` input of the Write Characters To File VI to True so that the new temperature values are appended to the selected file as the loop iterates. Click the Operating tool on the constant to set its value to True.

5. Return to the front panel and run the VI with the `# of points` set to 20. A file dialog box prompts you for a file name. When you enter a file name, the VI starts writing the temperature values to that file as each point is generated.
6. Save the VI, name it `My Write Temperature to File.vi`, and close the VI.
7. Use any word processing software such as Write for Windows, Teach Text for Macintosh, and Text Editor in Open Windows for UNIX to open that data file and observe the contents. You should get a file containing twenty data values (with a precision of six places after the decimal point) separated by commas.

Reading Data from a File

OBJECTIVE You will create a VI that reads the data file you wrote in the previous example and displays the data on a waveform graph. You must read the data in the same data format in which you saved it. Therefore, since you originally saved the data in ASCII format using string data types, you must read it in as string data with one of the file I/O VIs.

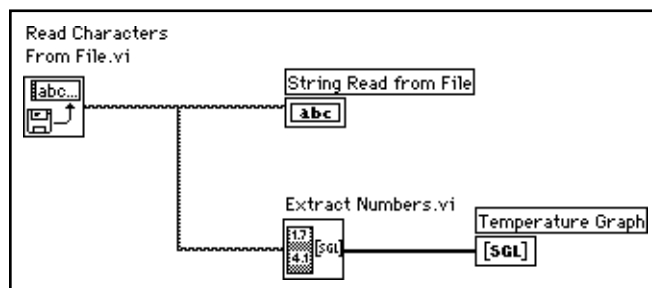
Front Panel



1. Open a new front panel and build the front panel shown in the preceding illustration.

The front panel contains a string indicator and a waveform graph. The String Read from File indicator displays the comma delimited temperature data from the file you wrote in the last example. The graph displays the temperature curve.

Block Diagram



1. Build the block diagram as shown in the preceding illustration.



The Read Characters From File VI (**Functions»File I/O**) reads the data from the file and outputs the information in a string. If no path name is specified, a file dialog box prompts you to enter a file name. In this example, you do not need to determine the number of characters to read because there are fewer characters in the file than the default (512).

You must know how the data was stored in a file in order to read the data back out. If you know how long a file is, you can use the Read Characters From File VI to determine the known number of characters to read.



The Extract Numbers VI (`examples\general\strings.llb`) takes an ASCII string containing numbers separated by commas, line feeds, or other non-numeric characters and converts them to an array of numerics.

2. Return to the front panel and run the VI. Select the data file you just wrote to disk when the file dialog box prompts you. You should see the same data values displayed in the graph as you saw in the My Write Temperature to File VI example.
3. Save the VI, name it `My Temperature from File.vi`, and close the VI.

Using the File I/O Functions

Sometimes the file I/O functions do not provide the functionality you may need for saving data to disk. At this point, you must use the functions from **Functions»File I/O»Advanced**.

Specifying a File

There are two ways to specify a file—programmatically or through a dialog box. In the programmatic method, you supply the filename and the pathname for the VI.

(Windows) A pathname consists of the drive name (for example, C), followed by a colon, followed by backslash-separated directory names, followed by the filename. An example is `C:\DATADIR\TEST1` for a file named TEST1 in the directory DATADIR on the C drive.

(Macintosh) A pathname consists of the drive name, followed by a colon, followed by colon-separated folder names, followed by the filename. An example is `HardDrive:DataFolder:Test1` for a file named Test1 in the folder DataFolder on the HardDrive.

(UNIX) A pathname consists of slash-separated directory names followed by the filename. An example is `/usr/datadirectory/test1` for a file named `test1` in the directory `/usr/datadirectory`.

(All Platforms) Using the dialog box method, the File Dialog function displays a dialog box that you can use to interactively search for a directory and then type in a filename.

Paths and Refnums



A path consists of a LabVIEW data type that identifies files. You can enter or display a file path using the standard syntax for a given platform with the path control and path indicator. In many ways, the path control and indicator works like a string control or indicator, except that LabVIEW formats the path appropriately for any platform supported by LabVIEW.



A refnum consists of a LabVIEW data type that identifies open files. When you open a file, LabVIEW returns a refnum associated with the file. All operations performed on open files use the file refnums to identify each file. A refnum is only valid for the period during which the file is open. If you close the file, LabVIEW disassociates the refnum with the file. If you subsequently open the file, the new refnum may be different from the refnum that LabVIEW used previously.

In addition to associating an operation with a file, LabVIEW remembers information for each refnum, such as the current location for reading from the file and the degree of access to the file that other users are permitted, so that you can have concurrent but independent operations on a single file. If you open a file multiple times, each open file operation returns a different refnum.

The File I/O functions do not contain any error checking or handling and merely return an error value. Therefore, when you build applications that use low-level functions, you must write your own error checking and handling to avoid any problems that may occur.

File I/O Examples

You can use the following examples to see how to use the File I/O functions complete with proper error checking and handling techniques:

The Write to Text File VI (in `examples\file\smp1file.llb`) writes an ASCII text file that contains data values with time-stamps.

The Read from Text File VI (in `examples\file\smp1file.llb`) reads an ASCII text file that contains data values with time-stamps.

Summary

A string is a collection of ASCII characters. String controls and indicators are located in **Controls»String & Table**.

LabVIEW contains many functions for manipulating strings. These functions are located in **Functions»String**.

LabVIEW can perform file operations. When writing to a file, you create a new file (or open an existing one), write the data, and close the file. Similarly, when you read from a file, you open an existing file, read the data, and close the file. For common file I/O operations, the file I/O VIs combine these steps into a single subVI. For greater flexibility, you can use the intermediate-level utility VIs or the File I/O functions.

Remember that you can use strings or other data types in file input and output operations. If the VI writes string data, it forms an ASCII file, while other forms of data produce a binary file. The binary file is faster and more compact, while the ASCII file is compatible with other programs and readable as text. See the following discussion on how to create and read binary data files.

Do not forget to use error checking and handling when writing data to or reading data from files. You can save time and effort by monitoring the error output values from the File I/O functions.

Additional Topics

Datalog Files

The examples shown in this chapter illustrate simple methods for dealing with files that contain data stored as a sequence of ASCII characters. This approach is common when creating files that other software packages read, such as a spreadsheet program. LabVIEW has another file format, called a *datalog file*. A datalog file stores data as a sequence of records of a single, arbitrary data type, which you determine when you create the file. LabVIEW indexes data in a datalog file in terms of these records. While all the records in a datalog file must be of a single type, that type can be complex. For instance, you can set each record so that the record contains a cluster with a string, a number, and an array.

If you are going to retrieve the data with a LabVIEW VI, you may not want to write data to ASCII files, because converting data to and from strings can be time consuming. For instance, converting a two-dimensional array to a string in a spreadsheet format with headers and time-stamps is a complicated operation. If you do not need to have the data stored in a format that other applications can read, you may want to write data out as a datalog file. In this form, writing data to a file requires little manipulation, making writing and reading much faster. It also simplifies data retrieval, because you can read the original blocks of data back as a log or record without having to know how many bytes of data the records contain. LabVIEW records the amount of data for each record of a datalog file.

The Write to Datalog File VI (in `examples\file\datalog.llb`) creates a new datalog file and writes the specified number of records to the file. Each record is a cluster containing a string and an array of single precision numbers.

To read a datalog file, you must match the data type that was used to write to the file. The Read from Datalog File VI (in `examples\file\datalog.llb`) reads a datalog file created by the Write to Datalog File example one record at a time. The record read consists of a cluster containing a string and an array of single precision numbers.

Binary Byte Stream Files

Writing data to binary byte stream files can be faster and can use less disk space than ASCII byte stream files. However, this method is typically more complicated because you must carefully design the file format and reconstruct the original data. If the file contains a number of different components of data written in binary format, and you want to retrieve the data in its original format, embed header information in the file that describes the structure of the data. For instance, before writing an array of numeric data to a file, the header information tells you how much data you need to reconstruct the original components.

The `Binary vs ASCII` example in `examples\general\strings.llb` shows the difference between a binary string and an ASCII string.

Refer to the file `I/O Write To I16 File, Read From I16 File, Write To SGL File, and Read From SGL File` (**Functions»File I/O»Binary File VIs**) for examples of writing to and reading from binary byte stream files.

Error I/O in File I/O Functions

File I/O functions also contain error I/O clusters, which consist of error in and error out to check for errors. With error I/O clusters you can string together several functions. When an error occurs in a function, that function does not execute and then passes the error along to the next function. For more information on error I/O, see **Online Reference»Function and VI Reference»Error I/O in File I/O Functions**.

Customizing VIs

You Will Learn:

- How to use the **VI Setup...** option.
- How to use the **SubVI Node Setup...** option.
- How to make custom controls and indicators.

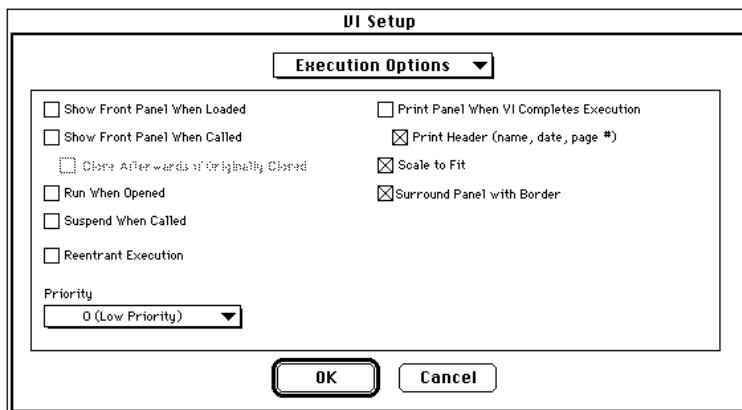
For examples of custom VIs, see `examples\general\viopts.llb`.

VI Setup

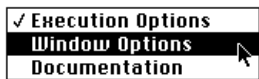
There are several ways you can set your VIs to execute. You access these options by popping up on the icon pane in the upper right corner of the front panel and choosing **VI Setup...**



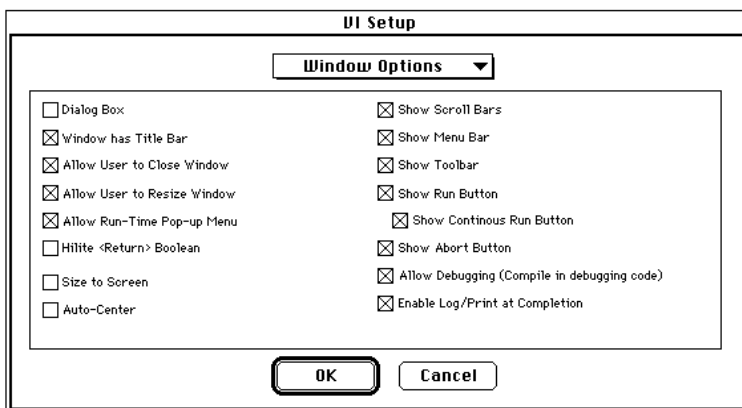
As the following illustration shows, a VI Setup dialog box appears showing all setup options. These options are described in detail in Chapter 6, *Creating Pop-Up Panels and Setting Window Features*, of the *LabVIEW User Manual*.



Setting Window Options

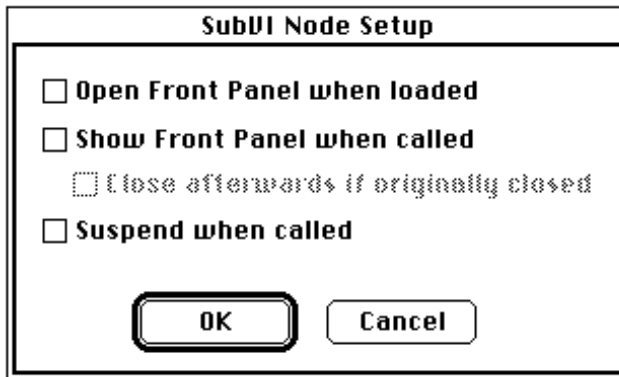


The Window Options control the appearance of the VI when running. To switch from **Execution Options** to **Window Options**, click on the downward pointing arrow in the menu bar.



SubVI Node Setup

There are several setup options on a subVI that you can modify. These options are available by popping up on the subVI icon (in the block diagram of the calling VI), and choosing **SubVI Node Setup....** The following illustration shows the SubVI Node Setup dialog box.



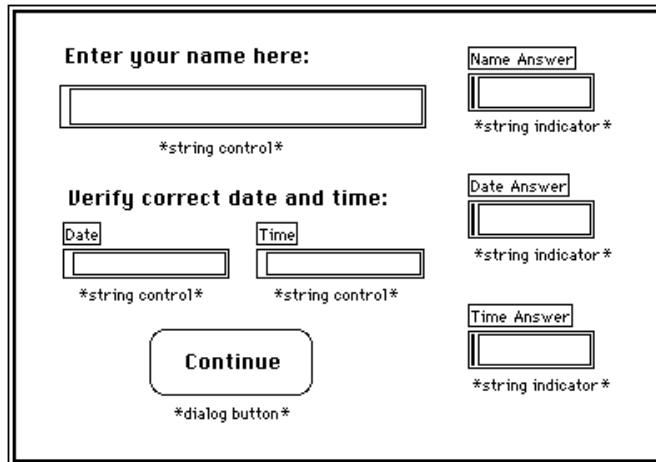
Note: *If you select an option from the VI Setup... dialog box of a VI, the option applies to every instance of that VI. If you select an option from the SubVI Node Setup dialog box, the option applies only to that particular node.*

Using Setup Options for a SubVI

OBJECTIVE You will build a VI that simulates a testing application. When the VI starts, a subVI opens a front panel and prompts the user to enter a name and to verify the date and time. The front panel remains open until the user clicks on **Continue**.

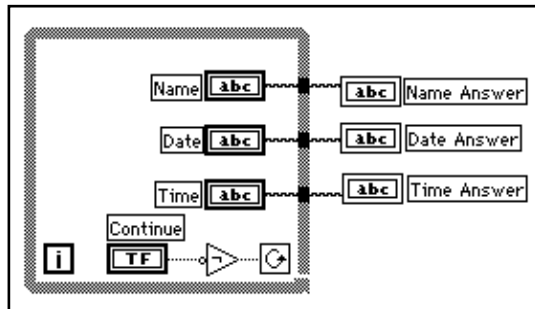
First, you must build a VI that pops open its front panel, prompts the user for information, and waits until the user clicks on a Boolean button. You then will use this VI as a subVI in the block diagram of the main VI.

Front Panel



1. Open a new front panel and build the front panel shown in the preceding illustration.

Block Diagram



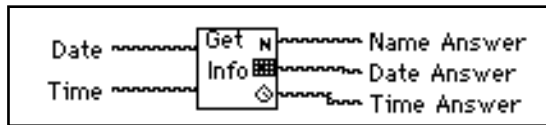
1. Build the block diagram shown in the preceding illustration.



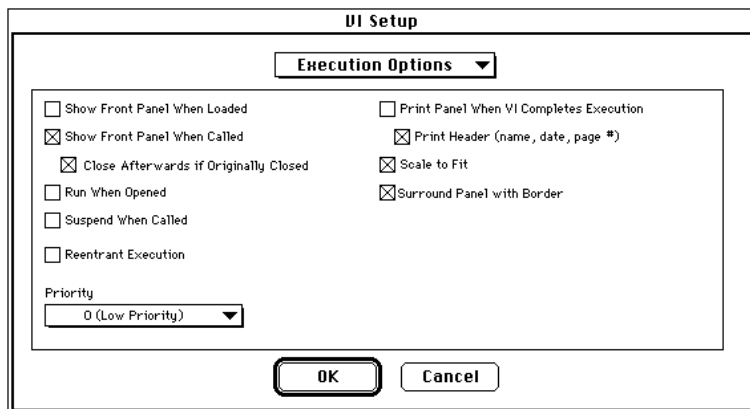
Not function (**Functions»Boolean**). In this exercise, the Not function inverts the value of the **Continue** button so that the While Loop executes repeatedly until you click on the **Continue** button. (The default state of the button is FALSE.)



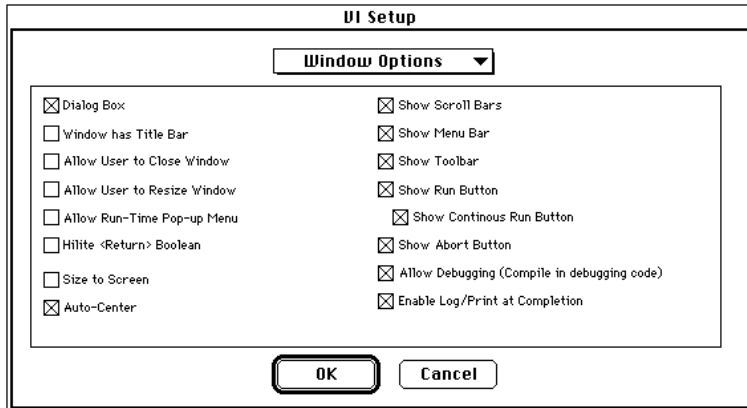
2. Create the icon for the VI as shown at left. To access the Icon Editor, pop up on the icon pane of the front panel and select **Edit Icon**.
3. Switch to the connector pane by popping up on the icon pane and selecting **Show Connector**.
4. Build the connector. When you build the connector, notice that the default connector pane is not what you see illustrated to the left. To get the correct connector pane, choose **Patterns** from the pop-up menu on the connector. Choose the pattern with two inputs and three outputs. Then choose **Flip Horizontal**. Now you can connect the Date and Time Controls to the two connectors on the left side of the icon, and the Name Answer, Date Answer, and Time Answer indicators to the three connectors on the right side of the icon, as illustrated by the following Help window. After you create the connector, return to the icon display.



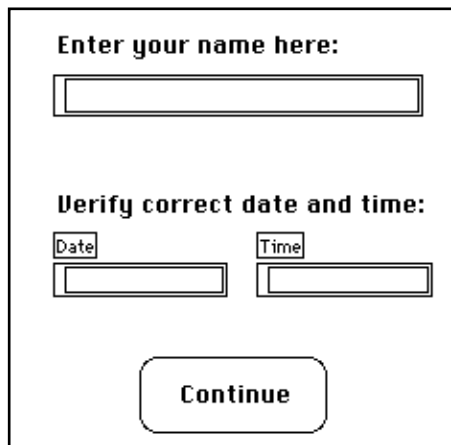
5. Save the VI as My Get Operator Info.vi. Now you can customize the VI with the VI setup options to make it look like a dialog box.



Configure the Execution Options as shown in the previous illustration. Then change to **Window Options** and configure it as shown in the following illustration.

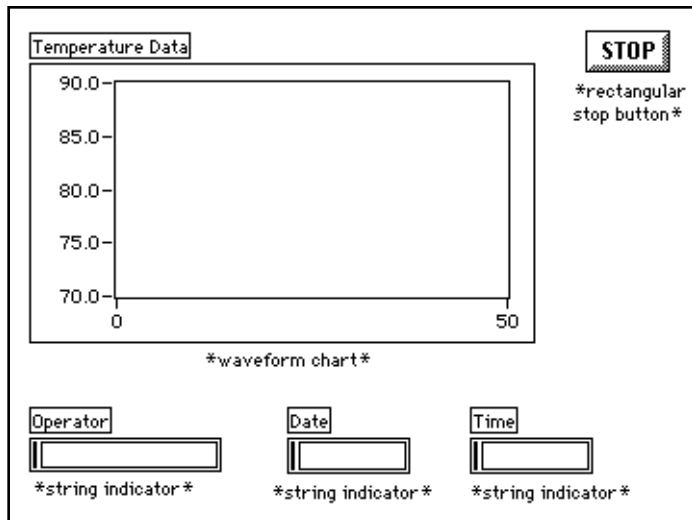


- After you finish with the VI Setup options, resize the front panel as shown in the following illustration so you do not see the three string indicators.



- Save the VI and close the VI. You will use this VI as a subVI soon.

Front Panel

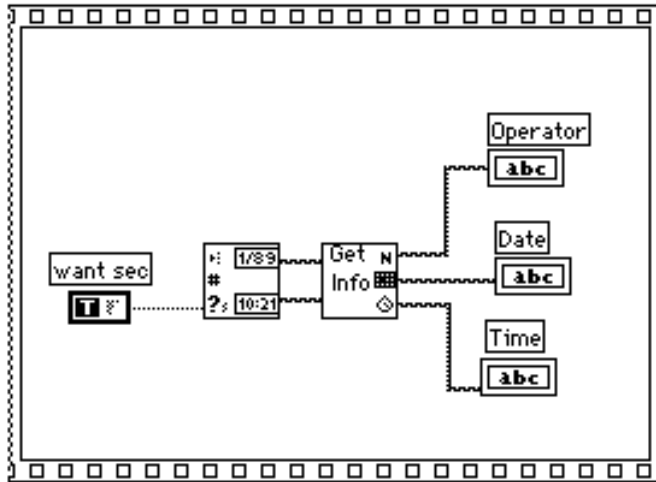


1. Open a new front panel.
2. Place a Waveform Chart (Controls»Graph) on the front panel and label it Temperature Data.
3. Modify the scale of the chart, so that its upper limit is set to 90.0 and its lower limit is set to 70.0.
4. Build the rest of the front panel as shown in the preceding illustration.
5. Pop up on the rectangular stop button and select **Mechanical Action»Latch When Released**.



The chart displays the temperature as it is acquired.

Block Diagram



6. Build the block diagram as shown in the preceding illustration.
7. Add a Sequence structure and add the following to frame 0.



The Get Date/Time String function (**Functions»Time & Dialog**) outputs the current date and time.



The Get Operator Info VI (**Functions»Select a VI...**) pops open its front panel and prompts the user to enter a name, the date, and the time.



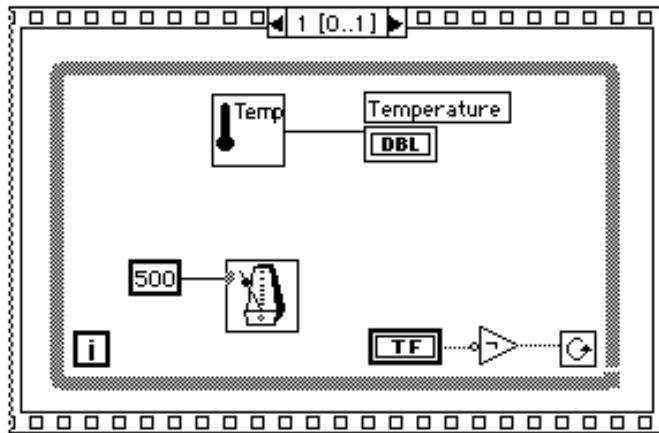
Boolean constant (**Functions»Boolean**) controls whether the input date and time string are true. To set this option to True, click on the constant with the Operating tool.

8. Pop up on the Sequence structure and select **Add Frame After** from the pop-up menu.



9. Place a While Loop inside frame 1 of the Sequence structure.

10. Add the objects shown in the following illustration to frame 1.



The Digital Thermometer VI (**Functions»Tutorial**). Returns one temperature measurement from a simulated temperature sensor or the My Thermometer VI (**Functions»Select a VI...**) you built in Chapter 2.



Wait Until Next ms Multiple function (**Functions»Time & Dialog**) causes the For Loop to execute in ms.



Numeric constant (**Functions»Numeric**). You can also pop up on the Wait Until Next Tick Multiple function and select **Create Constant** to automatically create and wire the numeric constant. The numeric constant delays execution of the loop for 500 ms (0.5 seconds).



Not function (**Functions»Boolean**) inverts the value of the Stop button so that the While Loop executes repeatedly until you click on Stop.

11. Save the VI. Name it My Pop-up Panel Demo.vi.

12. Run the VI. The front panel of the Get Operator Info VI opens and prompts you to enter your name, the date, and the time. Click on the Continue button to return to the calling VI. Then temperature data is acquired until you click on the STOP button.



Note:

The front panel of the Get Operator Info VI opens because of the options you selected from the VI Setup... dialog box. Do not try to open the front panel of the subVI from the block diagram of the My Pop-up Panel Demo VI.

13. Close all windows.

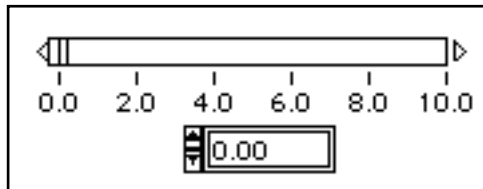
Custom Controls and Indicators

LabVIEW has a Control Editor that you can use to customize a front panel control. You can also use the Control Editor to save a customized control, so that you can use it in other LabVIEW VIs.



To invoke the Control Editor, select a control with the Positioning tool and then choose **Edit»Edit Control**.

When you edit a control, a new window opens with a copy of the control.



Now, you can customize the control by coloring it, changing its size, adding new elements to clusters, and so on. These changes do not affect the original VI until you select **File»Apply Changes**, or you close the window and select YES to the prompt concerning replacing the original control.

If you want to use the control in other VIs, you can save it as a *custom control* by selecting **File»Save**. After you save the control, you can place it on other front panels using the **Controls»Select a Control...**

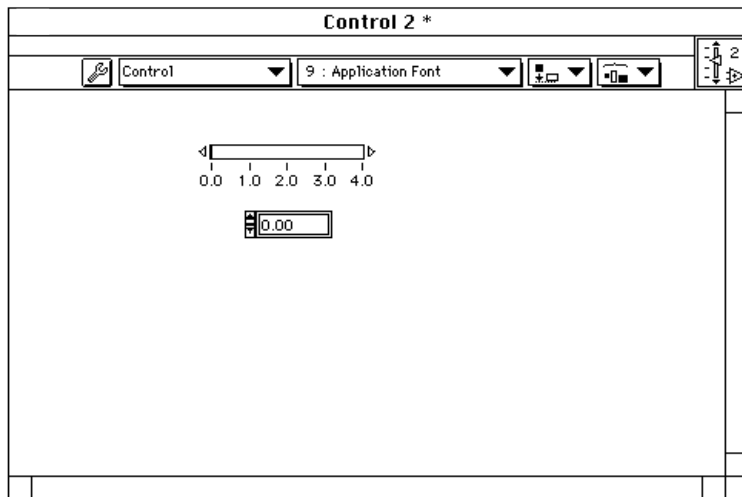
You can also import pictures from draw or paint programs into parts of a control or indicator. For example, assume you have a horizontal slide control and you want to make the slider look like a box and the housing look like a conveyor belt. You can load an example of this control by popping up in a front panel, selecting the **Controls»Select a Control...** option, and opening `examples\general\controls\custom.llb\box on belt`.

1. **(Windows and Macintosh)** Use a graphics program to draw a box and then import that image into LabVIEW through the clipboard.

(UNIX) Use a graphics program to draw a box and then save the image as an xwd (X Window Dump) file. Import the picture into the LabVIEW clipboard by selecting **File»Import Picture...**



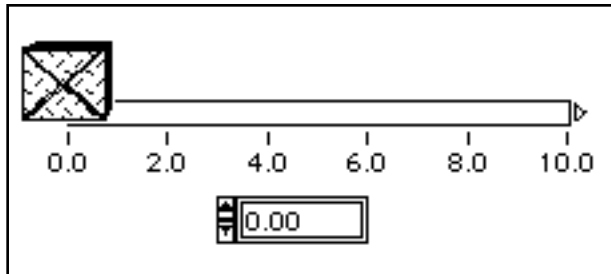
2. In LabVIEW, select the simple horizontal slide from **Controls»Numeric**.
3. Highlight the slide with the Positioning tool and select **Edit»Edit Control**. The following Editing window appears.



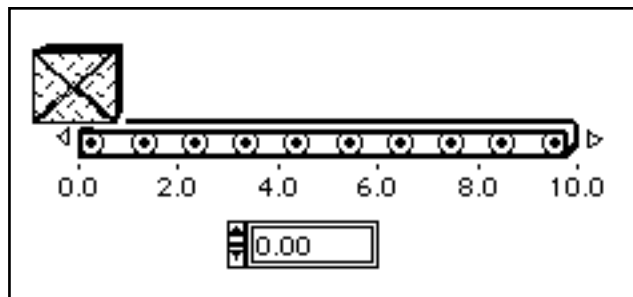
4. Use the Positioning tool to select the slide.
5. Click on the wrench, shown at left, in the Editing window toolbar.



6. Pop up on the slider in the horizontal slide and choose **Import Picture** to paste the picture onto the slider. The picture must be on LabVIEW's clipboard to perform this step. See step 1 for information on importing a picture to the clipboard. The box should replace the slide as in the following example.



7. Now draw the conveyor belt and import it into LabVIEW as in step 1. Return to the Editing window and repeat steps three through five. The horizontal slide should now look like the slide shown in the following illustration.



You can save this slide for use in other VIs. For more information regarding the Control Editor, see Chapter 23, *Custom Controls and Type Definitions*, in the *LabVIEW User Manual*.

For examples of custom controls, see `examples\general\control`.

Summary

With setup options, you can modify VI execution characteristics. These modifications include hiding the toolbar buttons, running the VI when loaded, opening front panels when called, and so on. You can modify options in two ways: using the VI Setup dialog box or using the SubVI Node Setup dialog box.

Any execution characteristic of a VI modified using the VI Setup dialog box affects *every* execution of that VI as a main VI or as a subVI. Any execution characteristic of a VI modified using the SubVI Node Setup dialog box affects only that node. Other nodes of the same VI are not affected.

The VI Setup dialog box also features options to hide the toolbar buttons. These options are useful when building VIs that other users operate, or building VIs for more complex test systems.

You can design your own controls or indicators by using the Control Editor. You can even import custom pictures into controls using the Control Editor.

Additional Topics

Simulating a Control/Indicator

In LabVIEW a front panel object is either a control or an indicator. In some cases, you may want a control that behaves as both a control and an indicator. That is what the previous example did with the date and time strings. You wanted to display the current date and time, but sometimes the internal clock of the computer is not correct. In this case, a control and an indicator are needed—an indicator so that the block diagram can set the date and time, and a control so that you can modify those values if they are incorrect.

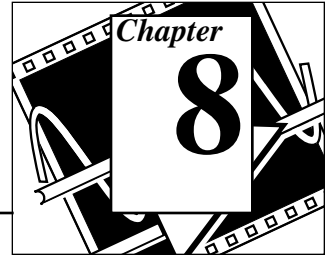
The My Pop-Up Panel VI served this purpose. The indicator from the main program became the control for the pop-up front panel. That way the user could modify the value and pass it out through an indicator and into the front panel.

Another way you can make a control/indicator pair is to use a local variable. A local variable acts as a multiple terminal to a front panel control or indicator. You can both read from and write to a local variable. To see examples of how to use local variables, see `examples\general\locals.llb`. For further information on local variables, see the *Local Variables* section, in Chapter 22, *Global and Local Variables*, in the *LabVIEW User Manual*.

Using the Control Editor

You can choose to save a custom control as a *type definition* or as a *strict type definition*. The Control Editor is useful in large applications, because you can change a saved control in one VI and it automatically updates in other VIs. For more information on type definitions, see the *Type Definitions* section in Chapter 23, *Custom Controls and Type Definitions*, in the *LabVIEW User Manual*.

Data Acquisition and Instrument Control



You Will Learn:

- About acquiring data with a plug-in data acquisition board (***Windows, Macintosh, and Sun***).
- About VISA functions.
- About GPIB functions.
- About serial port communication.
- About using a serial port to communicate with other serial ports.
- About using VXI for data acquisition (***Windows, Macintosh, and Sun***).
- What an instrument driver is.
- How to use a frequency response test VI.

Using LabVIEW to Acquire Data

One of the most valuable features of LabVIEW is its ability to acquire data from almost any source. LabVIEW contains VIs for controlling the following:

- Plug-in data acquisition boards (***Windows, Macintosh, and Sun***)
- GPIB (IEEE 488) instruments
- Serial port instruments
- VXI instruments (***Windows, Macintosh, and Sun***)

These VIs use the National Instruments industry-standard driver-level software to deliver complete control of your data acquisition and instrument control hardware.

This tutorial concentrates on basic LabVIEW features and functions. To learn more about LabVIEW data analysis capabilities, see Chapter 1, *Introduction to Analysis in LabVIEW*, in the *LabVIEW Analysis VI Reference Manual*.

About Plug-in Data Acquisition Boards (Windows, Macintosh, and Sun)

National Instruments manufactures all the components you need to build complete data acquisition systems. Plug-in boards are available for the IBM PC/AT, EISA, IBM PS/2/MicroChannel, Macintosh NuBus Series, Macintosh LC/LCII, and SPARCstation SBus computers.

These boards have various combinations of analog, digital, and timing inputs and outputs. You can use front-end SCXI signal conditioning multiplexers to cost-effectively increase the number of analog input channels. A wide variety of signal conditioning modules for thermocouples, resistance temperature detectors (RTDs), voltage and current inputs, and high current digital inputs and outputs complete the acquisition hardware line.

The LabVIEW data acquisition (DAQ) VIs control all of the National Instruments data acquisition hardware. To learn more about the LabVIEW DAQ library, see Chapter 2, *Installing and Configuring Your Data Acquisition Hardware*, of the *LabVIEW Data Acquisition Basics Manual*, which outlines the procedure for installing LabVIEW and configuring your system. The *LabVIEW Data Acquisition Basics Manual* also includes information to help you start building a data acquisition system with LabVIEW. For a description of the DAQ boards associated with various types of hardware, see Appendix B, *Hardware Capabilities in LabVIEW*, of the *LabVIEW Data Acquisition VI Reference Manual*.

For examples of DAQ VIs, see `examples\daq`.

About VISA

VISA is a single interface library for controlling GPIB, VXI, and other types of instruments. Using the VISA functions, you can construct a single instrument driver VI, which controls a particular instrument model across several different I/O media. A string is passed to the VISA Open function in order to select which type of I/O to use to communicate with the instrument. Once the session with the instrument is open, the VISA functions, such as VISA Read and VISA Write, perform the instrument I/O activities in a generic manner. Thus, the program is not tied to any specific GPIB or VXI functions. The

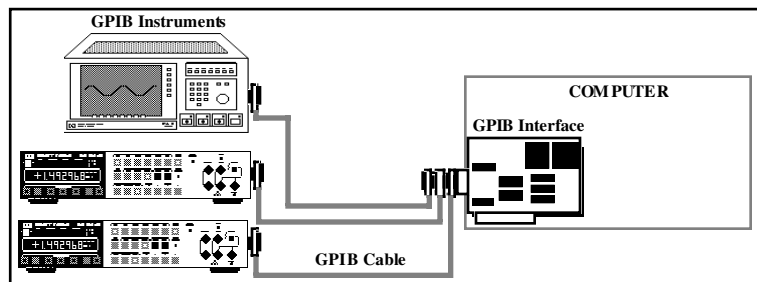
VISA instrument driver is considered to be interface independent and you can use in several different systems.

Instrument drivers that use the VISA functions, capture the activities specific to the instrument, not the communication medium. This can open more opportunities for reusing the instrument driver with a number of programs.

For examples of VISA functions, see `examples\instr`.

About GPIB

The General Purpose Interface Bus (GPIB), also called IEEE 488, is a method of communicating with stand-alone instruments, such as multimeters and oscilloscopes. National Instruments manufactures many products for controlling instruments with the GPIB. The most direct method is to install a plug-in GPIB board in your computer and connect your instrument directly to this board with a GPIB cable.



The LabVIEW GPIB functions control National Instruments GPIB interfaces. A description and history of the GPIB, or IEEE 488, is discussed in the *GPIB Overview* section, in Chapter 1, *Introduction*, of the *LabVIEW Instrument I/O VI Reference Manual*. LabVIEW uses the National Instruments standard NI-488.2 driver software that comes with your GPIB interface.

The GPIB library (**Functions»Instrument I/O**) contains both traditional GPIB functions and 488.2 functions. The GPIB 488.2 functions add IEEE 488.2 compatibility to LabVIEW. These functions implement calls that IEEE 488.2 specifies and resemble the routines in National Instruments NI-488.2 software.

For examples of GPIB functions, see `examples\instr`.



Note: *If possible, you should use the VISA function rather than GPIB because of VISAs versatility.*

About Serial Ports

Serial communication is a popular means of transmitting data between a computer and a peripheral device such as a printer, a plotter, or a programmable instrument. Serial communication uses a transmitter to send data, one bit at a time, over a single communication line to a receiver. This method of communication is common when transferring data at low rates or over long distances. For instance, serial data can be transferred via modems, over standard telephone lines.

Serial communication is popular because most computers have one or two serial ports. A limitation of serial communication, however, is that a serial port can communicate with only one device. To accommodate several devices, you must use a board with multiple serial ports or a serial port multiplexer box.

LabVIEW contains serial port VIs, which are described in Chapter 8, *Serial Port VIs* of the *LabVIEW Instrument I/O VI Reference Manual*. Before you begin using LabVIEW for serial communication, you should first make certain that the instrument is connected correctly to the computer. For Windows, you must also make certain that there are no interrupt conflicts. One way for Windows and Macintosh users to do this is to use a general terminal software program such as Microsoft Windows Terminal or ZTerm. Once you have established communication with an instrument, you are now ready to use the LabVIEW serial port VIs located in **Functions»Instrument I/O»Serial**.

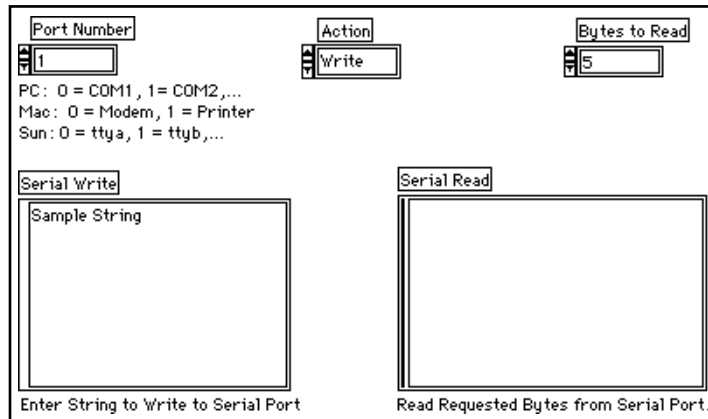
LabVIEW contains five VIs for serial communication—Serial Port Init, Serial Port Write, Serial Port Read, Bytes at Serial Port, and Serial Port Break. These functions are described in Chapter 8, *Serial Port VIs*, of the *LabVIEW Instrument I/O VI Reference Manual*.

For examples of serial port VIs, see `examples\instr\smplsrl11b`.

Using Serial Ports

OBJECTIVE You will examine a serial port example that you can use to communicate with any serial device. You will notice that serial communication is very similar to GPIB communication and frequently involves only writing and reading ASCII strings to and from a device.

Front Panel

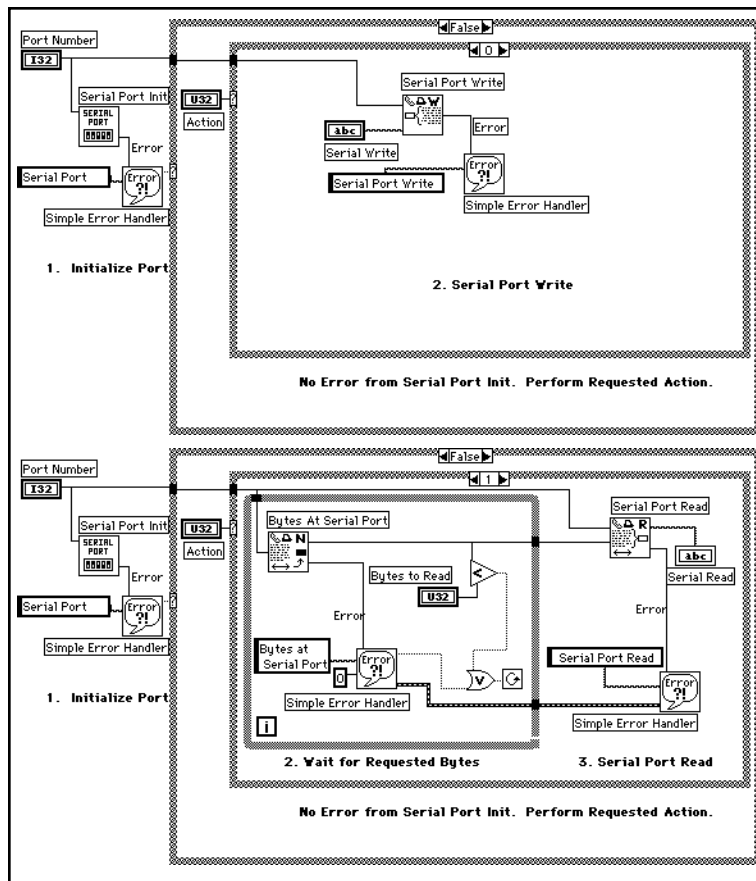


Open LabVIEW<->Serial.vi in the path `examples\instr\smp1ser1.llb`.

The general procedure to follow for serial port communication with LabVIEW starts with the Serial Port Init VI, shown at left. This VI sets the port number to use, the baud rate, the number of data bits and stop bits, and other parameters necessary for serial communication. Next, you use the Serial Port Write VI to send the necessary commands to the instrument for the operation you want it to perform.

To read information from a serial instrument, first run the Bytes at Serial Port VI. This VI checks how many bytes of information are waiting at the serial port. Usually, you use this VI in a loop to wait for a specified number of bytes to accumulate in the buffer before you read the information. Once you have the correct number of bytes at the serial port, you use the Serial Port Read VI to actually acquire the data.

Block Diagram



1. Open the block diagram of the LabVIEW<->Serial VI, some of which is shown in the preceding illustration, and observe the functions used to communicate via serial port in the method described previously. Click on the arrows at the top of the structures to examine the entire block diagram.



Serial Port Init function (**Functions»Instrument I/O» Serial**) initializes the selected serial port to the specified settings for baud rate, buffer size, data bits, stop bits, and parity.



Serial Port Write function (**Functions»Instrument I/O» Serial**) writes a data string to the indicated serial port. A serial command string usually consists of a group of ASCII characters.



Bytes at Serial Port function (**Functions»Instrument I/O» Serial**) returns the number of bytes in the input buffer of the serial port. Notice that this VI waits in a loop until the requested number of bytes is in the buffer. You may want to add a timeout to this loop in the case that the serial device never sends the requested number of bytes.



Serial Port Read function (**Functions»Instrument I/O» Serial**) reads the specified number of characters from the serial port.



Simple Error Handler function (**Functions»Time & Dialog**) informs the user if an input error exists, describes the error, and identifies where it occurred. See the discussion in the *Additional Topics* section, at the end of this chapter, for more information about error handling.

About VXI for Windows, Macintosh, and Sun

The VXIbus is a fast-growing platform for instrumentation systems. VXI uses a mainframe chassis with a maximum of thirteen slots to hold modular instruments on plug-in boards. A variety of instrument and mainframe sizes are available from numerous vendors, and you can use multiple instrument sizes in the same mainframe. You can control a VXI mainframe in several different ways.

LabVIEW has VXI VIs for high- and low-level control of a VXI system. You access these VIs from **Functions»Instrument I/O»VISA**. For more information on how to acquire data and control instruments with VXI, see the *LabVIEW Instrument I/O VI Reference Manual*.

About Instrument Drivers

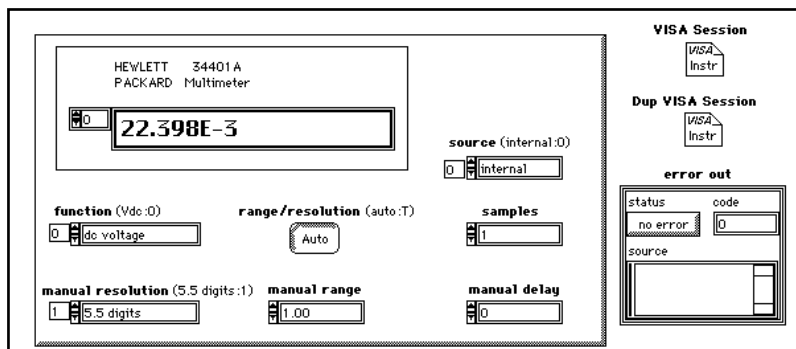
An instrument driver is software that controls a particular instrument. LabVIEW is ideally suited for creating instrument drivers. A LabVIEW front panel can simulate the operation of a front panel of an instrument. The block diagram can send the necessary commands to the instrument to perform the operation the front panel specifies. When you finish building an instrument driver, you no longer need to remember the commands necessary to control the instrument. Rather, you need only specify the input on the front panel. There is little value in having a software panel to control the instrument. The real value is that you can use the instrument driver as a subVI in conjunction with other subVIs in a larger VI to control an entire system.

LabVIEW has a library of over 500 instrument drivers for GPIB, serial, CAMAC, and VXI (for Windows, Macintosh, and Sun) instruments. Because there are many different types of instruments, it is impossible to demonstrate the techniques for creating drivers for all types of instruments; however, for a message-based instrument all drivers build a *command string* and send it to the instrument to perform the operation that the simulated front panel specifies. The command string consists of device-specific commands (usually in ASCII) that remotely control the instrument. Therefore, instrument drivers contain more string manipulation functions than specific interfacing commands. For more information about instrument drivers, see Chapter 3, *Developing a LabVIEW Instrument Driver*, of the *LabVIEW Instrument I/O VI Reference Manual*.

Using Instrument Drivers

OBJECTIVE You will examine the instrument driver for the Hewlett Packard 34401A Multimeter. Although this instrument driver is written for GPIB control, remember that the main component of an instrument driver is string manipulation.

Front Panel



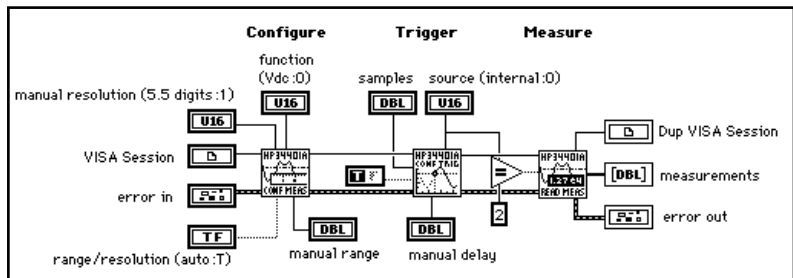
1. Open the HP34401A Application Example.vi, located in `labview\examples\instr\hp34401a.llb`.

The front panel contains several controls for configuring the multimeter for different measurements.

The VISA Session input identifies the device with which the VI communicates and passes all necessary configuration information required to perform I/O. You must run the HP34401A Initialize VI to establish communication with the instrument and to obtain the VISA Session value.

Notice also that the Error Out cluster describes any error that may have been generated by the VI during execution.

Block Diagram

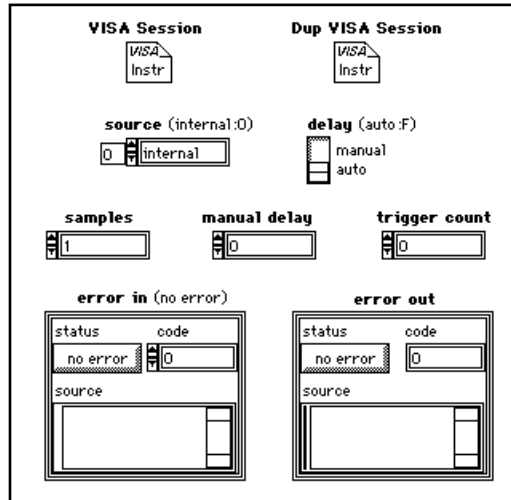


1. Open the block diagram of the HP34401A Application Example.vi.

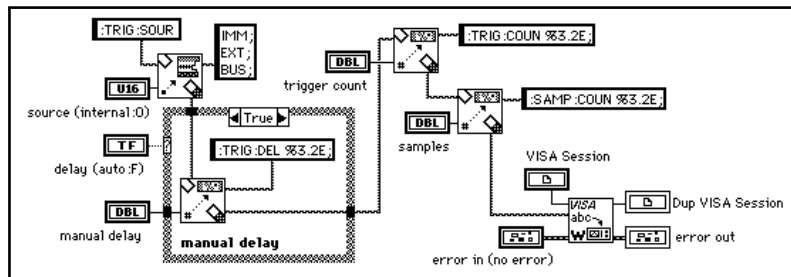
The block diagram calls three subVIs: the HP34401A Config Measurement subVI, the HP34401A Config Trigger subVI, and the HP34401A Read Measurement subVI.

The instrument drivers for LabVIEW consist of several VIs that you can use to control the instrument and an example that shows how to use those VIs. Typically, each VI should perform a specific task such as configuration, triggering setup, or reading a measurement. All the VIs were created in LabVIEW, so you can modify the code for your application to combine several tasks or to take out some functionality for increased performance.

- Open the HP34401A Config Trigger subVI by double-clicking on it.



- Switch to the block diagram, and examine how the instrument driver was written.



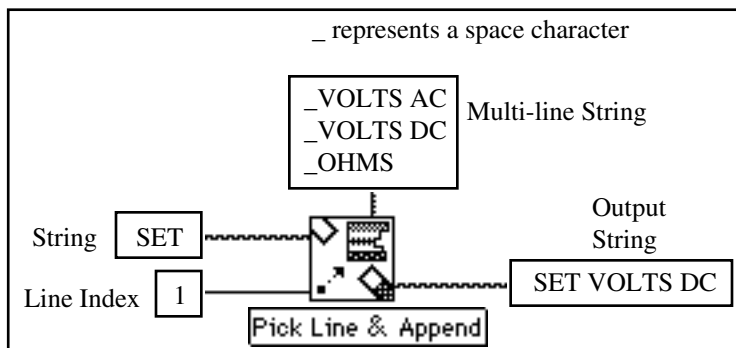
Notice that the instrument driver mainly consists of string manipulation functions for building the command strings for the instrument. This example uses VISA Write, which sends the command string to GPIB or VXI message-based instruments.

The following functions are commonly used to write instrument drivers.



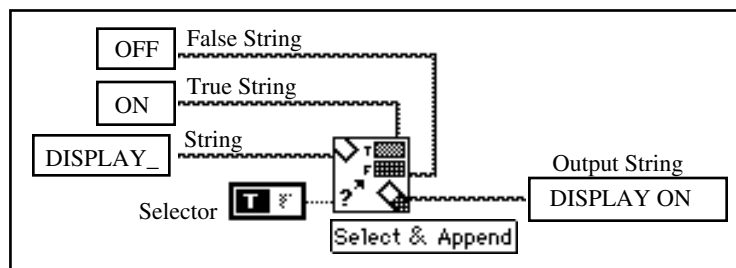
Pick Line & Append function (**Functions»String**) chooses a line from a multiline string and appends that line to a string.

In the following example, the function selected the string VOLTS DC and appended it to the string SET. Notice that Line Index 1 chooses the second line, because the first line index is zero.



Select & Append function (**Functions»String**) chooses a string according to a Boolean selector and appends that string to the output string.

In the following example, the VI appends the true string ON to the input string DISPLAY.



Match Pattern function (**Functions»String**) searches a string for a specified expression and returns the matched string, the string before the match, and the string after the match. Match Pattern is a very powerful function; for more information on Match Pattern, see Chapter 4, *String Functions*, in the *LabVIEW Function Reference Manual*.

4. When you finish examining the instrument driver, close the VI and do not save any changes that you have made.

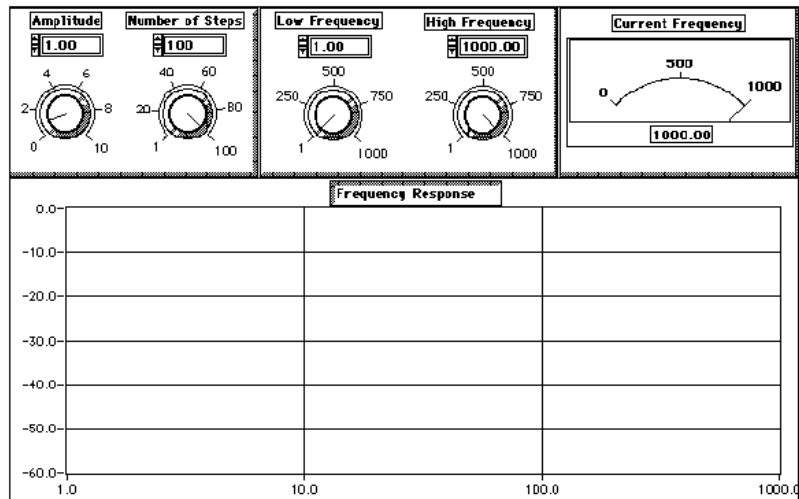
A large number of instrument drivers exist for LabVIEW. If the instrument you are using does not have an instrument driver VI, you can take an instrument driver that was written for a similar instrument or by the same instrument manufacturer and modify the command strings to match your particular instrument.

Using a Frequency Response Test VI

Typically, instrument drivers like the one you examined in the last example are used for test and measurement applications where several instruments are used. Imagine an application that uses GPIB instruments to perform a frequency response test on a unit under test (UUT). A function generator supplies a sinusoidal input to the UUT and a digital multimeter measures the output voltage of the UUT. You would then like to examine the resulting voltage response curve on an XY Graph.

OBJECTIVE You will use a VI that simulates using GPIB instruments to perform a frequency response test on a UUT as described in the preceding paragraph.

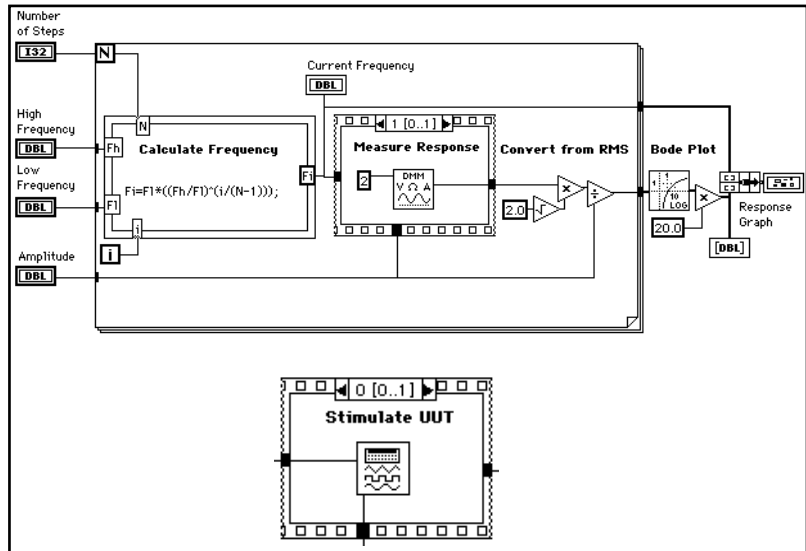
Front Panel



1. Open `Frequency Response.vi`, located in `examples\apps\freqresp.lib`.

The front panel contains several knobs for configuring the function generator. You can set the amplitude, low frequency, high frequency, and the number of frequency steps to take. A meter displays the current frequency of the function generator, and an XY Graph displays the frequency response plot.

Block Diagram



1. Open the block diagram of the Frequency Response VI.

The block diagram contains a For Loop at the highest level. The loop executes N times specified by the Number Of Steps knob.

A Formula Node takes the values specified by the Low Frequency, High Frequency, and Number Of Steps knobs to calculate the i th frequency such that all N frequencies are equally spaced on a log axis. Once the Formula Node calculates the i th frequency value, the node passes the frequency value and the value specified by the Amplitude knob to a Sequence structure.

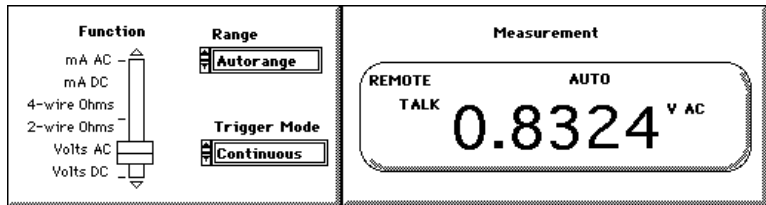


Frame 0 contains the Demo Tek FG 5010 subVI. It is built much like the Fluke 45 driver that you examined earlier. The Demo Tek FG5010 subVI node needs only two input parameters: frequency and amplitude. The driver default function (sine wave) and mode (continuous signal) are correct for this application and do not need to be changed.

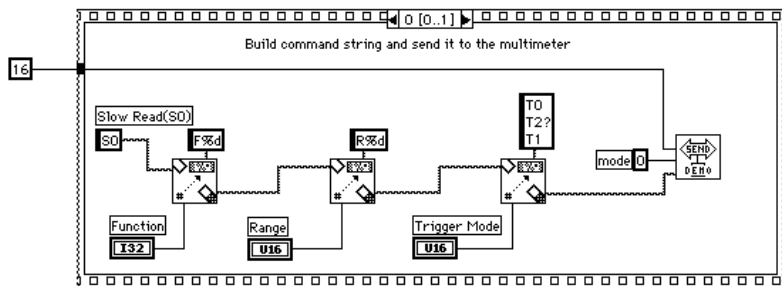


Frame 1 contains the Demo Fluke8840A subVI.

- Open the DemoFluke8840A VI by double-clicking on it with the Operating tool. Notice that the front panel, shown in the following illustration, was designed to look like the actual instrument.



- Open the block diagram, part of which is shown in the following illustration, and examine how the instrument driver was written. Notice that it is almost identical to the instrument driver you examined before for the Fluke 45 multimeter.



- When you finish examining the demo instrument driver, close the VI and do not save any changes you may have made.
- Finish examining the block diagram of the Frequency Response VI.

The DemoFluke8840A subVI returns an RMS voltage, which is converted to a peak-to-peak voltage by multiplying it by the square root of 2. A Bode plot of the frequency response plots the UUT gain in dB versus the frequency $.20 \log$ (peak-to-peak voltage converts the UUT gain to dB).

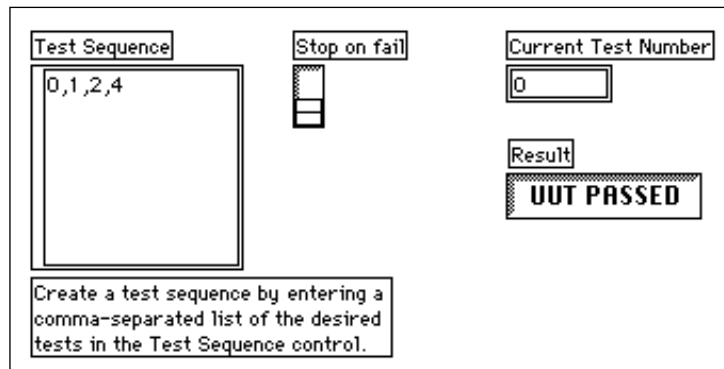
Each tunnel at the edge of the For Loop automatically assembles the values from each iteration into an array by using the auto-indexing feature described in Chapter 4, *Arrays, Clusters, and Graphs*, of this tutorial. The program then uses a Bundle function to combine the x and y arrays into the graph.

- Return to the front panel and run the VI. Change the knobs to see different frequency response plots.
- Close the Frequency Response VI and do not save any changes.

Writing a Test Sequencer

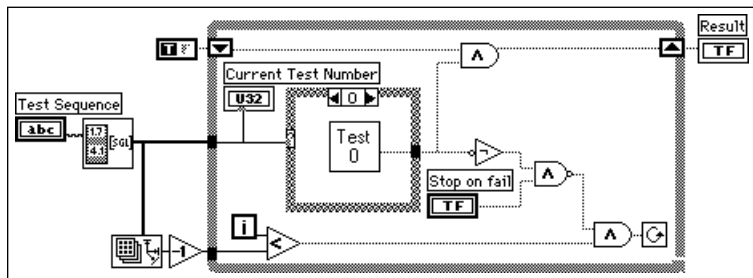
Many production and Automated Test Executive (ATE) applications like the one you just examined require the automated sequencing of tests. Each test measures a particular characteristic of the unit under test (UUT) and compares that measurement to an expected limit. If the value is within the limit, the test passes. Otherwise, the test fails. A complete testing procedure for a UUT consists of a number of these tests executed sequentially. The following example shows how to build a test sequencer.

Front Panel



- Open `Test Sequencer.vi`, located in `examples\apps\testseq.llb`.

Block Diagram



1. Open the block diagram of the Test Sequencer VI as shown in the preceding illustration.

The test sequence consists of a string that the VI expands into a numeric array of tests to run. This array is passed to a While Loop that contains a Case structure. Each case contains a test VI that corresponds to a number in the test sequence string. The While Loop indexes through the array of test numbers, selecting the appropriate case index for each test. The Current Test Number indicator shows the test number currently executing. If a test fails and the operator has set Stop on fail to TRUE, the loop stops.

In this example, each test passes out a Boolean value that indicates whether or not the test passed. The Boolean is set to TRUE if the test passed and FALSE if it fails. The main test sequencer loop uses a shift register to keep a cumulative PASS/FAIL status for the entire testing sequence. When the loop stops executing, the test sequence displays the cumulative status in the Result indicator.

There are many extensions that you can make to this basic test sequencer. One common modification would be to store test sequences in files rather than display them in a front panel control. The operator could then load the specific test sequence that is appropriate for the current UUT. Some other common extensions include:

- Generating a text report of test results.
- Prompting the operator for a UUT serial number.
- Looping a test on failure.

You can add all of these capabilities to the basic test sequencer described in this example. However, you may want to obtain the LabVIEW Test Executive Toolkit, which contains a full-featured, test sequencer that implements not only the previously mentioned extensions, but also test dependencies, force conditions, and more. Contact National Instruments for purchasing information regarding the LabVIEW Test Executive Toolkit.

Summary

There are many ways to acquire data with LabVIEW. GPIB is a useful method for communication with individual instruments. Serial port communication is a useful method for transferring data over long distances.

(Windows, Macintosh, and Sun) You can use VIs to control plug-in data acquisition boards and VXI systems.

(Windows, Macintosh, and Sun) You can learn more about using the data acquisition VIs for acquiring data with plug-in data acquisition boards from the *LabVIEW Data Acquisition Basics Manual*.

(All Platforms) Just as the GPIB VIs eliminate the need to have an in-depth knowledge of the IEEE 488 bus, the VIs in the LabVIEW Instrument Driver library eliminate the need to have an in-depth knowledge of a specific instrument. There are over 300 instrument drivers in the library. If you have an instrument that is not on the list, you can find a similar instrument and modify that instrument driver for your instrument.

VISA functions are the preferred method for controlling instruments, such as GPIB, VXI, and so on. Because VISA functions concentrate on the activities specific to the instrument, VISA controlled instrument drivers are considered to be interface independent.

The GPIB functions control GPIB communication. The commonly used functions are GPIB Write, GPIB Read, and GPIB Status. The GPIB Write function sends data to an instrument. The GPIB Read function reads data from an instrument. The GPIB Status VI returns the status of the GPIB when you execute the VI. Several other GPIB functions perform less common operations such as clearing a device, waiting for a service request, triggering a device, or polling a device.

You can control serial devices with the serial port VIs. There are only five serial port VIs. To control a serial device you use the Serial Port Init VI to configure the communication port. Then, you use the Serial Port Write VI to send a command string. You can wait until a set number of characters are in the buffer with the Bytes at Serial Port VI. Finally, you read the data from the buffer with the Serial Port Read VI.

(Windows, Macintosh, and Sun) You can control VXI systems if you have the LabVIEW VXI Development System. For more information refer to the *LabVIEW VXI VI Reference Manual*.

LabVIEW has many string functions ideally suited for instrumentation programming. These functions help you easily convert data from one type to another, or extract numbers from strings.

When acquiring data and controlling instruments, you may have a particular test suite that controls your application. You can use the Test Sequencer example to build such a test suite. The LabVIEW Test Executive Toolkit is available as an add-on package if you want to have full control of your ATE application. Contact National Instruments for more information on the LabVIEW Test Executive Toolkit.

Additional Topics

Error Handling

You should use error checking whenever possible when developing applications with I/O operations. LabVIEW contains three error handler utilities, which are described in the *LabVIEW User Manual*. These VIs consist of the Simple Error Handler VI, the Find First Error VI, and the General Error Handler VI. You can connect these VIs to error status terminals of other VIs to test whether an error has occurred. If an error has occurred, these VIs return a text-based description of the error. In addition, you can use these VIs to display a dialog box containing a description of the error message.

These error handlers not only contain the error messages from all the GPIB and serial port functions, but they also contain error messages for all the file I/O and analysis operations.

Waveform Transfers

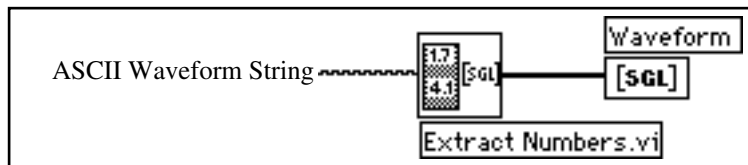
Most GPIB digitizers, such as oscilloscopes and scanners, return a waveform as either an ASCII string or a binary string. Assuming the same waveform, a binary string transfer would in most cases be faster and require less memory than an ASCII string transfer. This is because binary encoding usually requires fewer bytes than ASCII encoding.

ASCII Waveforms

As an example, consider a waveform composed of 1,024 points, each point having a value between 0 and 255. Using ASCII encoding, you would need a maximum of 4 bytes to represent each point (a maximum of 3 bytes for the value of the point and 1 byte for the separator, such as a comma). You would need a maximum of 4,096 (4 * 1,024) bytes plus any header and trailer bytes to represent the waveform as an ASCII string. The following illustration shows an example of this ASCII waveform string.

CURVE {12,28,63,...1024 points in total...}CR LF		
Header	Data Point	Trailer
(6 bytes)	(up to 4 bytes each)	(2 bytes)

You can use the Extract Numbers VI (from `examples\general\string.llb`) to convert an ASCII waveform into a numeric array, as the following illustration shows.

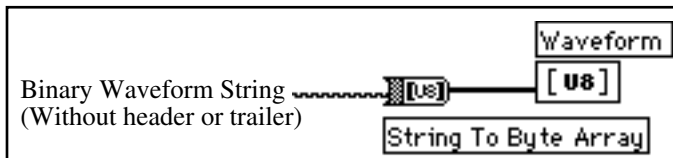


Binary Waveforms

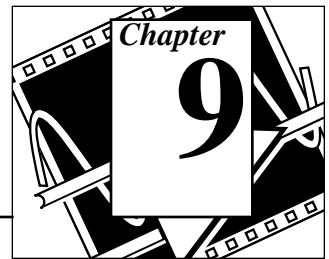
The same waveform using binary encoding would require only 1,024 bytes (1 * 1,024) plus any header and trailer bytes to be represented as a binary string. Using binary encoding, you would need only 1 byte to represent the point, assuming each point was an unsigned 8-bit integer. The following illustration shows an example of a binary waveform string.

<code>CURVE</code>	<code>%</code>	<code>{MSB}</code>	<code>{LSB}</code>	<code>{\hat{A} \hat{a}...1024 bytes in total...}</code>	<code>{Chk}</code>	<code>CR LF</code>
Header		Count	DataPoint			Trailer
(7 bytes)		(4 bytes)	(1 byte each)			(3 bytes)

Converting the binary string to a numeric array is a little more complex. You must convert the string to an integer array. You can do this by using the String To Byte Array function found in **Functions»String»Conversion**. You must remove all header and trailer information from the string before you can convert it to an array. Otherwise, this information is also converted.



Programming Tips and Debugging Techniques



You Will Learn:

- Tips for programming VIs.
- Techniques for debugging VIs.

Programming Tips

The following tips contain helpful suggestions and techniques for programming VIs in LabVIEW.

Tip 1 Frequently used menu options have equivalent command key short cuts. For example, to save a VI, you can choose **File»Save**, or press <Ctrl-s> (Windows); <command-s> (Macintosh); <meta-s> (Sun); or <Alt-s>.

Action	Windows	Macintosh	Sun	HP-UX
Saves a VI.	Ctrl-s	command-s	meta-s	Alt-s
Runs a VI.	Ctrl-r	command-r	meta-r	Alt-r
Toggles between the front panel and block diagram.	Ctrl-e	command-e	meta-e	Alt-e
Toggles the Help window on/off.	Ctrl-h	command-h	meta-h	Alt-h
Removes all bad wires.	Ctrl-b	command-b	meta-b	Alt-b

Action	Windows	Macintosh	Sun	HP-UX
Lists all errors for a VI.	Ctrl-l	command-l	meta-l	Alt-l
Closes the active window.	Ctrl-w	command-w	meta-w	Alt-w

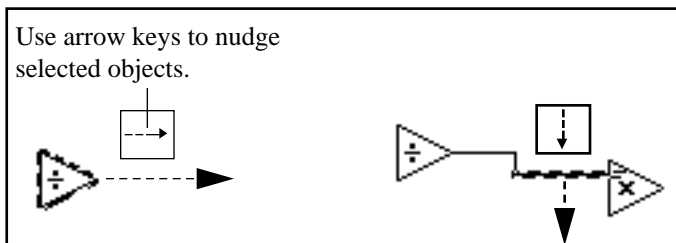
Tip 2 To rotate through the most commonly used tools in the **Tools** palette, press <Tab>.

Tip 3 To toggle between the Operating and Positioning tools on the front panel, press the spacebar. On the block diagram, the spacebar toggles between the Positioning and the Wiring tools.

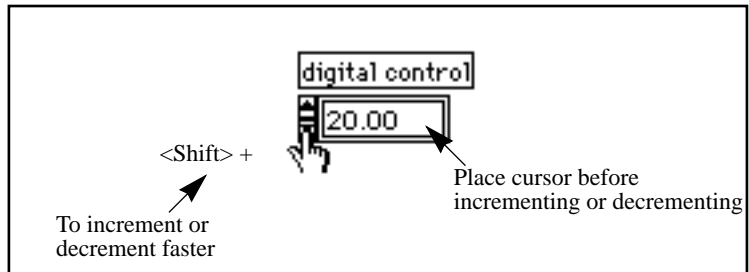
Tip 4 To convert any **Controls** or **Functions** palettes or subpalettes into floating palettes, push in the thumb tack, located in the upper-left corner of the palette.

Tip 5 To change the direction of a wire while wiring, press the spacebar.

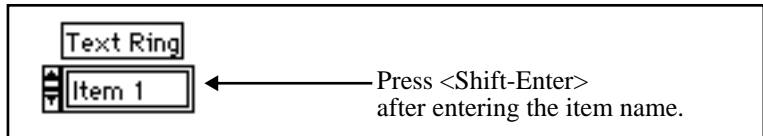
Tip 6 To move selected objects in the front panel and the block diagram, press the arrow keys. Pressing the arrow keys moves a selected object one pixel in the direction of the arrow. This tip also works for selected wire segments.



Tip 7 To increment or decrement faster, press <Shift> while you click the increment or decrement button on digital controls.



Tip 8 To add items to ring controls, press <Shift-Enter> (Windows); <shift-return> (Macintosh); <Shift-Return> (Sun); or <Shift-Enter> (HP-UX) after typing the item name. Pressing these keys accepts the item and positions the cursor to add the next item.

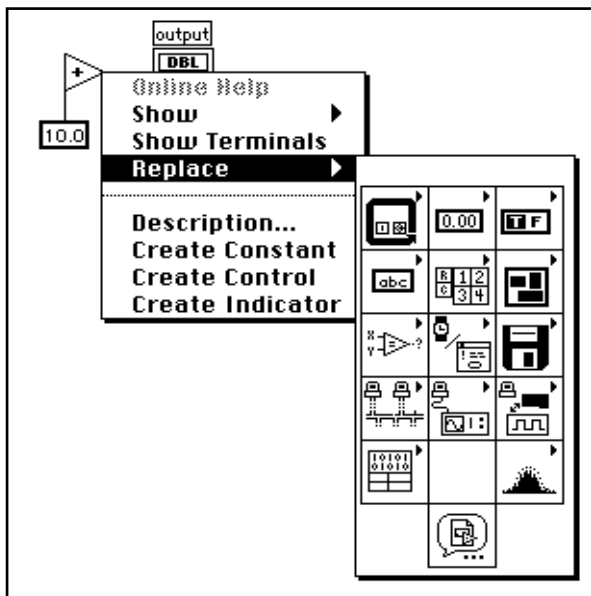


Tip 9 To duplicate an object, select the object using the Positioning tool, hold down <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX), and drag the mouse.



Tip 10 To limit an object to horizontal or vertical motion only, hold down <Shift> and drag the object with the Positioning tool.

Tip 11 To replace nodes, pop up on the node and choose **Replace** from the pop-up menu.



Tip 12 To pick a color from an object, first select the Color tool. Place the tool over the object and press and hold down <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX). The tool changes to the Color Copy tool. Pick up the object color by clicking on the object. Release the key and color other objects by clicking on them using the Coloring tool.



Tip 13 One common error you should avoid is wiring together two controls or wiring two controls to an indicator. This mistake returns the error message: Signal: has multiple sources. To fix this problem, pop up on the control and select **Change to Indicator**.

Tip 14 To automatically create and wire the correct constant, control, or indicator type to an object, pop up on the input or output of the object and select **Create Constant**, **Create Control**, or **Create Indicator**.

Tip 15 (*Windows and UNIX*) To delete a wire as you are wiring, right-click the mouse button.

Debugging Techniques

Finding Errors



When your VI is not executable, a *broken* arrow, shown at left, appears on the run button, in the toolbar. To list the errors, click on the broken run button. Click on one of the errors listed to highlight the object or terminal that reported the error.



Note:

The error list window also displays warnings if it is open, although warnings do not cause the run arrow to break. (Overlapping or partially hidden nodes and terminals are examples of warnings that may indicate the presence of a more serious design problem in your VI.)

Single Stepping Through a VI



For debugging purposes, you may want to execute a block diagram node-by-node. To enable the single step mode, click on the step over button, located in the toolbar.



To enable stepping over a loop, subVI, and so on, click on the step over button, located in the toolbar.



To enable stepping into a loop, subVI, and so on, click on the step into button, located in the toolbar.



To enable stepping out of a loop, VI, subVI, and so on, click on the step out button, located in the toolbar. You can specify how far you want the VI to execute before pausing by clicking on the step out button and holding the mouse button down. This accesses a pop-up menu.

Execution Highlighting



You can animate block diagram execution highlighting by clicking on the hilite execute button, located in the toolbar.



The symbol changes to the symbol shown to the left. You can use execution highlighting with single stepping to trace the flow of data in the block diagram.

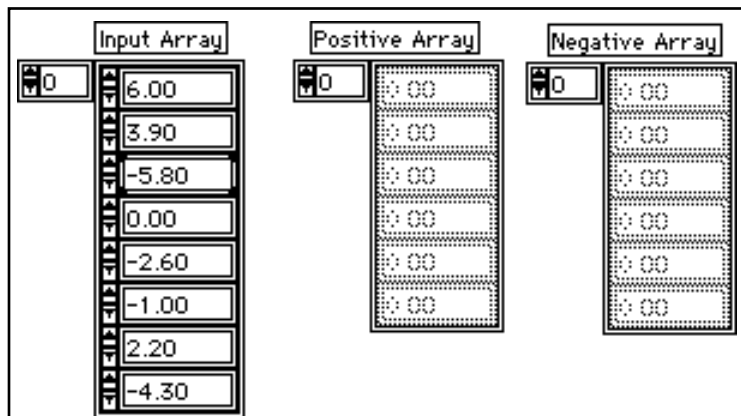
Debugging a VI

OBJECTIVE You will open a VI that checks each number in an array to see if it is negative. If the number is negative, the VI takes that number out of the array and places it in another array. In other words, the VI separates an input array of mixed positive and negative numbers into an array of positive numbers and an array of negative numbers.

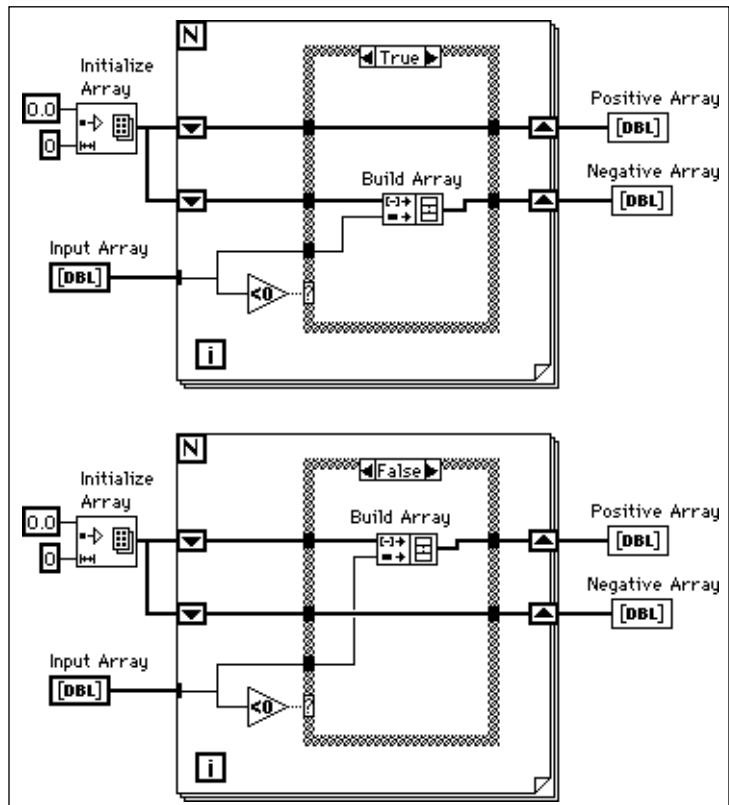
Front Panel

1. Open the *Separate Array Values.vi* by selecting **File>Open...**. The VI is located in `examples\general\arrays.llb`.

The array of digital controls supplies the input. The Positive Array and Negative Array indicators display the result of the separation.



Block Diagram



1. Open and examine the block diagram. You can display only one case at a time. To change cases, click on the arrows in the top border of the Case structure.

Notice that the count terminal of the For Loop does not have a specific value wired to it, and that the input array is auto-indexed as it enters the For Loop. The For Loop continues to run until there are no more elements in the array.

A good way to debug a VI is to single-step through the VI and animate the flow of data through the block diagram.



2. Enable the execution highlighting mode by clicking on the highlight execute button in the toolbar of the block diagram. The button changes from the symbol shown at the top-left to the symbol shown at the bottom-left.
3. Enable the single-step mode by clicking on the step over button, located in the toolbar. The VI passes the Input Array to the For Loop. The VI then passes the floating point and integer values of the numeric constants to Initialize Array.



Note:

Moving dots represent the dataflow in the block diagram. The number of elements in the various arrays are noted and specific values appear as the block diagram executes.



4. Click on the step over button. This initializes the shift registers.
5. Click on the step into button to step into the For loop. Values from the Initialize array function pass through the shift registers and then pause at the Case structure. The less than function determines whether the value from the input array, for this iteration, the value is six, is less than zero and then pauses execution.
6. Click on the step over button to pass the TRUE or FALSE result of the less than to the Case structure. For this iteration, the input value six is greater than zero, so that the Less Than function passes a result of FALSE to the Case structure.
7. Click on the step out button. This executes the rest of the block diagram as well as the subsequent iterations. Notice that the elements do not pass out of the For Loop to the indicators yet. When each iteration of the loop finishes, the VI transfers the resulting positive and negative arrays to the indicators. This is an important concept of dataflow programming—a loop does not begin to execute until all the data going into it is available and data does not leave the loop until the loop finishes execution.

LabVIEW includes a Breakpoint tool, which you can use to set breakpoints on nodes, block diagrams, structure objects, and wires.



Note:

Red frames around nodes and diagrams and red dots on wires indicate set breakpoints.



8. Select the Breakpoint tool from the **Tools** palette.
9. Place the Breakpoint cursor on the Initialize Array node and click on the node. A red frame appears around the Initialize Array node.



Note: *Make certain that the arrow on the Breakpoint cursor is pointing towards the structure or wire that you wish to set the breakpoint on.*

10. Single step through the VI by clicking on the step into button in the toolbar. LabVIEW highlights the Initialize Array node and stops execution right before the Initialize Array node executes.
11. Click the Breakpoint cursor on the Initialize Array node to remove the breakpoint.

LabVIEW also contains a Probe tool, which you can use to view the data as it flows through a wire.



12. Select the Probe tool from the **Tools** palette.
13. Place a probe on the wire connecting the Input Array to the Case structure. A probe window, labeled Probe 1, pops up and a yellow glyph with the number one appears on the wire. The probe window appears on both the front panel and block diagram.
14. Single-step through the VI again. The probe window displays the data value, consisting of 6.0 for this iteration, as it flows through that portion of the wire's segment.
15. Turn off the execution highlighting by clicking on the hilite execute button.



The button changes to the symbol shown to the left.



16. Close the VI by selecting **File»Close**. Do not save changes to the VI.

Opening the Front Panels of SubVIs

Another debugging technique is to open the front panels of subVIs to watch the data as it passes through each subVI. For example, if you have an application that contains a subVI to acquire data and then another subVI to analyze that data, you can open the front panels of both subVIs, and then run the main application. The run button of the subVI changes to indicate that the subVI is currently running. You can then verify that the acquire subVI actually reads the correct data and whether the analysis subVI obtains that data and calculates the appropriate output. The subVI controls and indicators change as

current values pass into the subVI. You can turn on execution highlighting and single stepping in the subVI if you see a potential problem and wish to examine it closer.

If you want to examine the subVIs of an application, be sure to open the front panels of the subVIs before you start the application. Otherwise, the subVIs contain their default values and you do not see the current values.

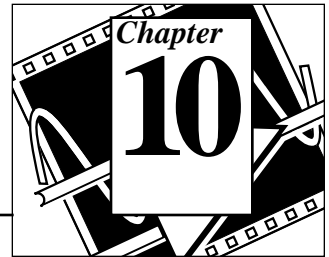
Summary



A broken arrow in the run button, located in the toolbar, identifies nonexecutable VIs. Clicking on the broken run button opens an Error List dialog box that lists the VI's errors. Execution highlighting and the single stepping helps you to debug your VIs easily by tracing the flow of data through the VI. The Breakpoint tool is useful for debugging because you can use it to pause VI execution at a specific point in the VI. The Probe tool is useful for debugging because it shows you the values in a wire as the VI runs.

If you feel a subVI is not working correctly, open its front panel before you run the main VI and watch which values are passed to and from that subVI.

Program Design



Congratulations! You have completed the tutorial, and are familiar with many aspects of the LabVIEW programming process. Now, you need to apply that knowledge to develop your own applications.

How do you start?

This chapter attempts to answer that question by suggesting some techniques to use when creating programs and offering programming style suggestions.

Use Top-Down Design

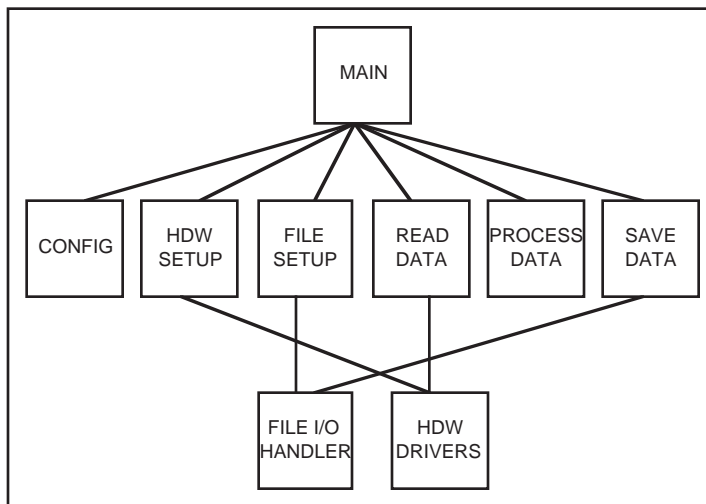
When you have a large project to manage, think *top-down* design. LabVIEW has an advantage over other programming languages when it comes to top-down design because you can easily start with the final user interface and then animate it.

Make a List of User Requirements

List types of I/O, sampling rates, need for real-time analysis, data presentation, and so on. Next, create some dummy front panels that you can show to the prospective users (or manipulate yourself, if you are the user). Think about and discuss functions and features. Use this interactive process to redesign the user interface, as necessary. You may need to do some low-level research at this early stage to be sure that you can meet specifications, such as data throughput.

Design the VI hierarchy

Break the task, at logical places, into manageable pieces. As the following flowchart shows, there are several major blocks that you can expect to see in one form or another on every data acquisition system.



In some cases you may not need all of these blocks, or you may need different blocks. For instance, in some applications you may not need to use any file I/O operations. Alternatively, you may need additional blocks, such as blocks representing user prompts. The main objective is to divide your programming task into these high-level blocks that you can easily manage.

After you determine the high-level blocks you need, try to create a block diagram that uses those high-level blocks. For each block, create a new *stub VI* (a nonfunctional prototype representing a future subVI). Give this stub VI an icon and create a front panel that contains the necessary inputs and outputs. You do not have to create a block diagram for this VI yet. Instead, see if this stub VI is a necessary part of your top-level block diagram.

After you assemble a group of these stub VIs, try to understand, in general terms, the function of each block, and how that block provides the desired results. Ask yourself whether any given block generates information that some subsequent VI needs. If so, make sure that your top-level block diagram sketch contains wires to pass the data between VIs.

Try to avoid using *global variables*, because they hide the data dependency between VIs. As your system gets larger, it becomes difficult to debug if you depend upon global variables as your method for transferring information between VIs.

Write the Program

You are now ready to write the program in LabVIEW.

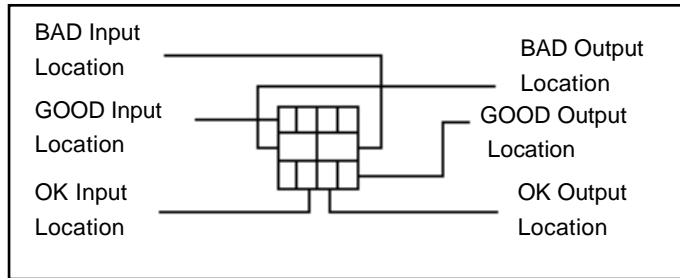
- Use a modular approach by building stub subVIs where there is a logical division of labor, or the potential for code reuse.
- Solve the more general problems along with your specific ones.
- Test your stub subVIs as you write them. This may involve construction of higher-level test routines, but it is much easier to catch the bugs in one small module than in a hierarchy of 75 VIs.

As you consider the details of your stub subVIs, you may find that your initial design is incomplete. For example, you may realize that you need to transfer more information from one subVI to another. You may have to reevaluate your top-level design at this point. Using modular subVIs to accomplish specific tasks makes it easier to manage your program reorganizations.

Plan Ahead with Connector Patterns

If you think that you may need to add additional inputs or outputs later on, select a connector pattern with extra terminals. You can leave these extra terminals unconnected. That way, you do not have to change the connector pattern for your VI if you find you need another input or output later on. Changing patterns requires replacement of the subVI in all calling VIs. By adding extra, unused terminals, you can add an input or output with minimal effect on your hierarchy.

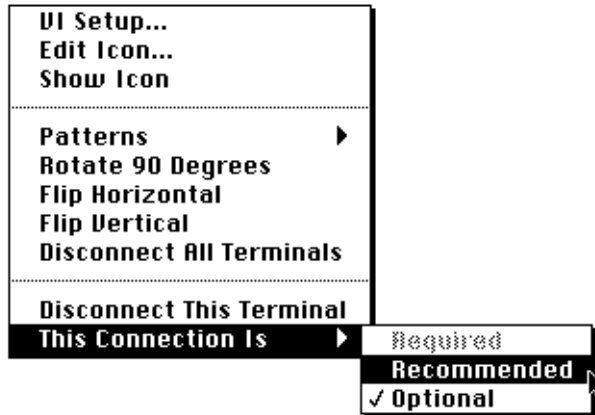
When linking controls and indicators to the connector, place inputs on the left, and outputs on the right. This prevents complicated, unclear wiring patterns in your VIs.



If you create a group of subVIs that are commonly used together, try to give the subVIs a consistent connector pattern, with common inputs in the same location. This makes it easier to remember where to locate each input without using the Help window. If you create a subVI that produces an output that is used as the input to another subVI, try to align the input and output connections. This simplifies your wiring patterns.

SubVIs with Required Inputs

On the front panel, you can edit required inputs for subVIs by clicking on the icon pane, on the upper-right side of the window and choosing **Show Connector»This Connection Is**. From the submenu, choose between the **Required**, **Recommended**, or **Optional** options. The following illustration displays the submenu options.



If you want to return to the icon pane in the front panel, pop up on the connector pane and select **Show Icon**.

Good Diagram Style

Avoid Oversized Diagrams

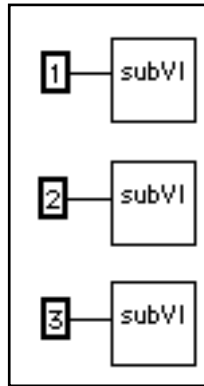
In general, avoid creating a block diagram that uses more than a page or two of screen space. If a diagram becomes very large, decide whether some components of your diagram could be reusable by other VIs, or whether a section of your diagram fits together as a logical component. If so, you should consider breaking your diagram up into subVIs.

With forethought and careful planning, it is much easier to design diagrams that use subVIs to perform specific tasks. Using subVIs helps you to manage changes and to debug your diagrams quickly. You can determine the function of a well-structured program after only a brief examination.

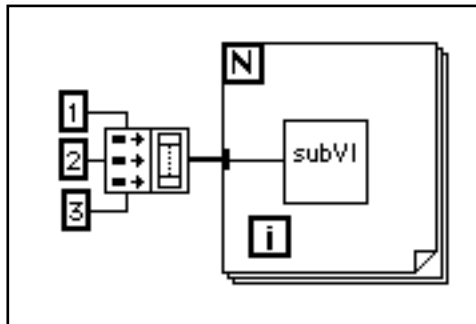
Watch for Common Operations

As you design your programs, you may find that you perform a certain operation frequently. Depending upon the situation, this may be a good place to use subVIs or loops to repetitively perform an action.

For example, consider the following diagram, where three similar operations run independently.



An alternative to this design is a loop, which performs the operation three times. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the loop.



If the array elements are constant, you can use an array constant instead of building the array on the block diagram.

Use Left-to-Right Layouts

LabVIEW was designed to use a left-to-right (and sometimes top-to-bottom) layout. All elements of your program should be organized in this fashion, when possible.

Check for Errors

When you perform any kind of I/O, you should consider the possibility of errors occurring. Almost all I/O functions return error information. Make sure that your program checks for errors and you handle them appropriately.

LabVIEW does not handle errors automatically, because users usually want very specific error-handling methods. For example, if an I/O VI in your block diagram times out, you may or may not want your entire program to halt. You also may want the VI to retry for a certain period of time. In LabVIEW, you make error-handling decisions.

The following list describes three situations in which errors frequently occur:

- Incorrect initialization of communication or data that has been improperly written to your external device
- Loss of power, broken, or improperly working external device
- Bugs in LabVIEW or other programs that occur when you upgrade LabVIEW or your system operating software

When an error occurs, you may not want certain subsequent operations to take place. For instance, if an analog output operation fails because you specify the wrong device, you may not want LabVIEW to perform a subsequent analog input operation.

One method for managing such a problem is to test for errors after every function, and put subsequent functions inside case structures. This can complicate your diagrams and ultimately hide the purpose of your application.

An alternative approach, which has been used successfully in a number of applications and many of the VI libraries, is to incorporate error handling in the subVIs that perform I/O. Each VI can have an error input and an error output. You can design the VI to check the error

input to see if an error has previously occurred. If there is an error, the VI can be set up to halt execution and to pass the error input to the error output. If there is no error, the VI can execute the operation and pass the result to the error output.

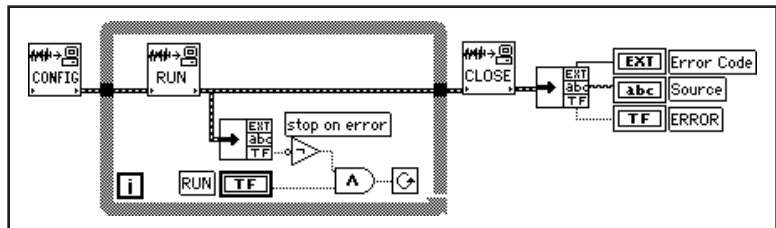


Note: *In some cases, such as a Close operation, you may want the VI to perform the operation regardless of the error that is passed in to it.*

Using the preceding technique, you can easily wire several VIs together, connecting error inputs and outputs to propagate errors from one VI to the next. At the end of series of VIs, you can use the Simple Error Handler VI to display a dialog box if an error occurs. The Simple Error Handler VI is located in **Functions»Time & Dialog**. In addition to encapsulating error handling, you can use this technique to determine the order of several I/O operations.

One of the main advantages in using the error input and output clusters is that you can use them to control the execution order of dissimilar operations.

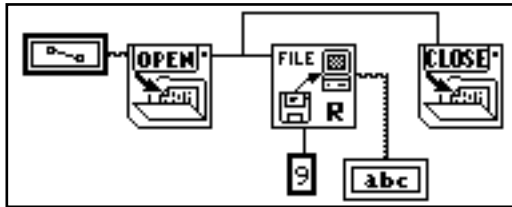
The error information is generally represented using a cluster containing a numeric error code, a string containing the name of the function that generated the error, and an error Boolean for quick testing. The following illustration shows how you can use this in your own applications. Notice that the While Loop stops if it detects an error.



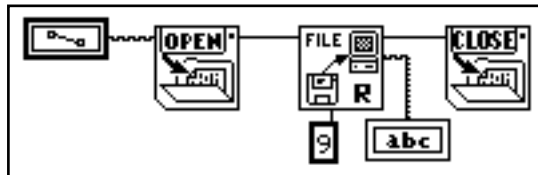
Watch Out for Missing Dependencies

Make sure that you have explicitly defined the sequence of events, when necessary. Do not assume left-to-right or top-to-bottom execution when no data dependency exists.

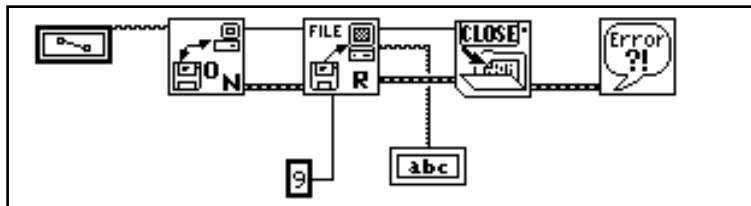
In the following example, there is no dependency between the Read File and the Close File. More than likely, this program cannot work as expected.



The following version of the block diagram establishes a dependency by wiring an output of the Read File to the Close File; the operation cannot end until the Close File receives the output of the Read File.



Notice that the preceding example still does not check for errors. For instance, if the file does not exist, the program does not display a warning. The following version of the block diagram illustrates one method for handling this problem. In this example, the block diagram uses the error I/O inputs and outputs of these functions to propagate any errors to the simple error handler VI.



Avoid Overuse of Sequence Structures

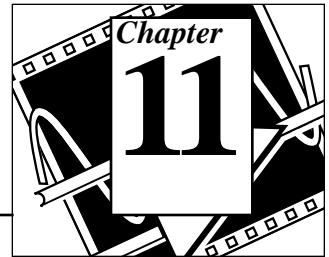
Because LabVIEW operates with a great deal of inherent parallelism, avoid overusing Sequence structures. Using a Sequence structure guarantees the order of execution, but prohibits parallel operations. For instance, asynchronous tasks that use I/O devices (GPIB, serial ports, and data acquisition boards) can run concurrently with other operations, if Sequence structures do not prevent them from doing so.

Sequence structures tend to hide parts of the program and interrupt the natural left-to-right flow of data. You pay no performance penalty for using Sequence structures; however, when you need to sequence operations, you might want to consider using dataflow instead. For instance, in I/O operations you might use the error I/O technique described previously to ensure that one I/O operation occurs before the other.

Study the Examples

For further information on program design, you can examine the many example block diagrams included in LabVIEW. These sample programs can provide you with insights into program style and building. To view these block diagrams, open the readme VI located in the `examples` directory. With this VI, you can access the numerous programming examples.

Where to Go from Here



The tutorial that you have just completed should prepare you to create LabVIEW applications. Before you start, you may want to examine some additional resources.

The `examples` directory contains a number of useful examples illustrating good programming techniques. At the top-level of the directory is a VI called `readme.vi`. With this VI, you can view the available examples. When you select a VI, LabVIEW displays the documentation for that VI (this information was previously entered for the VI using the VI Information dialog box). To open a VI, select **File>Open...**

(Windows, Macintosh, and Sun) The DAQ examples directory (for Macintosh, located in the `examples\daq` folder) contains a VI library called `RUN_ME` that has a Getting Started example VI for analog input, analog output, digital I/O, and counter/timers. The *LabVIEW Data Acquisition Basics Manual* contains information for these functional areas that guides you through the `RUN_ME` example VI and explains how the data acquisition VIs work. For information on how to use the same example VIs with SCXI hardware, see Part 5, *SCXI—Getting Your Signals in Great Condition*, in the *LabVIEW Data Acquisition Basics Manual*. The `RUN_ME` examples and the *LabVIEW Data Acquisition Basics Manual* provide an excellent starting place for information on data acquisition programming.

Other Useful Documentation

(Windows, Macintosh, and Sun) If you intend to use data acquisition in your program, you should read Chapter 3, *Basic LabVIEW Data Acquisition Concepts*, of the *LabVIEW Data Acquisition Basics Manual*. This chapter contains important information about using the data acquisition VIs with LabVIEW.

(All Platforms) The *LabVIEW User Manual* contains a number of chapters that describe advanced programming concepts. These concepts are not necessary for many applications, but can be very helpful if you plan to write large, LabVIEW applications. These chapters include discussions of custom controls and type definitions, performance tuning, and portability issues. The chapters also contain information that helps you to manage applications and understand how LabVIEW executes VIs. In addition, there is a helpful discussion about printing and documentation.

Chapter 1 of the *LabVIEW Communications VI Reference Manual*, discusses the options for networking in LabVIEW (TCP/IP, DDE, Apple Events, and PPC).

The *LabVIEW Cross Reference Manual* contains a comprehensive index to all of the LabVIEW manuals, a master glossary, and a complete listing of error codes.

For Information on Advanced Topics

This tutorial attempts to teach you the fundamentals of LabVIEW programming. LabVIEW contains some advanced features that are either not discussed or only discussed in a limited fashion in this tutorial. You should be aware of these features so that you can apply them as necessary in your applications.

The attribute node is described briefly in this tutorial. With the attribute node, you can programmatically manage settings related to controls and indicators. For example, you can change the visibility of controls using the attribute node. You can use the attribute node if you need to programmatically change the options in a ring or list control, clear the contents of a chart, or change the scales on a chart or graph. Attribute nodes are discussed in detail in Chapter 21, *Attribute Nodes*, in the *LabVIEW User Manual*.

This tutorial briefly discussed local variables. You can use local variables if you need to read from controls in multiple locations of your block diagram. They are also useful if you need to treat a front panel object as a control in some locations and an indicator in other locations, so that you can write to it and read from it on the block diagram. Local variables should be used judiciously, because they hide the data flow of your diagrams, which makes it difficult to see the purpose of your

program and to debug local variables. See Chapter 22, *Global and Local Variables*, of the *LabVIEW User Manual* for a discussion of local variables. Notice that applications that use local variables may make more copies of data than applications that do not; for a discussion of this, see Chapter 27, *Performance Issues*, also in the *LabVIEW User Manual*.

You can use global variables if you need to store data used by several of your VIs. Global variables should also be used judiciously, for the same reason as local variables. Global variables are necessary in some applications. However, do not use them if you can structure your program so that you can use an alternate data flow method for transferring data. See Chapter 22, *Global and Local Variables*, of the *LabVIEW User Manual* for details.

You can create subVIs from a selection on the block diagram using **Edit»Create SubVI from Selection**. In addition, LabVIEW automatically wires the correct inputs and outputs to the subVI. In some instances, you cannot create a subVI from a VI. See Chapter 4, *Creating SubVIs*, of the *LabVIEW User Manual* for a detailed discussion of this feature.

You can use the VI profile feature (**Project»Show Profile Window**) to access detailed information about a VIs timing statistics and timing details. This feature should help you to optimize the performance of your VIs. See Chapter 27, *Performance Issues*, of the *LabVIEW User Manual* for a detailed discussion of the profile feature.

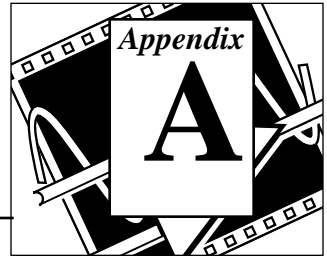
This tutorial briefly discussed the Control Editor. You can use the Control Editor to customize the look of your controls. You can also use the editor to save customized controls so that you can reuse them in other applications. See Chapter 23, *Custom Controls and Type Definitions*, of the *LabVIEW User Manual* for a detailed discussion of the Control Editor.

The list and ring controls are useful if you need to present the user with a list of options. See Chapter 14, *List and Ring Controls and Indicators*, in the *LabVIEW User Manual* for a detailed discussion of these controls.

LabVIEW has a Call Library function that you can use to call a shared library or DLL. With this function, you can create a calling interface in LabVIEW if you need to call an existing code or driver. See Chapter 24, *Calling Code from Other Languages*, in the *LabVIEW User Manual* for a discussion of the Call Library functions.

You can use code interface nodes (CIN), as an alternative method for calling source code written in a conventional, programming language from LabVIEW block diagrams. CINs are useful for tasks that conventional programming languages can perform more quickly than LabVIEW, tasks that you cannot perform directly from the block diagram, and for linking existing code to LabVIEW. However, the call Library function is generally easier to use when calling source code than CINs. You should use CINS when you need tighter integration with LabVIEW and the source code. See Chapter 24, *Calling Code from Other Languages*, in the *LabVIEW User Manual* for a discussion of CINs.

Customer Communication



For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a FaxBack system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services



Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422 or (800) 327-3077

Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 1 48 65 15 59

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity



FTP Support

To access our FTP site, log on to our Internet host, <ftp.natinst.com>, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.



FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following numbers:

(512) 418-1111 or (800) 329-7177



E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPiB: gpib.support@natinst.com

LabVIEW: lv.support@natinst.com

DAQ: daq.support@natinst.com

HiQ: hiq.support@natinst.com

VXI: vxi.support@natinst.com

VISA: visa.support@natinst.com

LabWindows: lw.support@natinst.com

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.



Telephone



Fax

Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	519 622 9311
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 48301892	02 48301915
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

Fax (____) _____ Phone (____) _____

Computer brand _____ Model _____ Processor _____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed _____

Hard disk capacity _____MB Brand _____

Instruments used _____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

List any error messages: _____

The following steps reproduce the problem: _____

Documentation Comment Form

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

Title: LabVIEW Tutorial Manual

Edition Date: November 1995

Part Number: 320998A-01

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name _____

Title _____

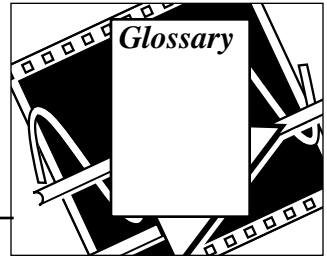
Company _____

Address _____

Phone () _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678



Prefix	Meaning	Value
m-	milli-	10^{-3}
μ -	micro-	10^{-6}
n-	nano-	10^{-9}

A

- absolute path** Relative file or directory path that describes the location relative to the top of level of the file system.
- active window** Window that is currently set to accept user input, usually the frontmost window. For Macintosh, the window is located on the desktop. The title bar of an active window is highlighted. You make a window active by clicking on it, or by selecting it from the Windows menu.
- ANSI** American National Standards Institute.
- array** Ordered, indexed set of data elements of the same type.
- array shell** Front panel object that houses an array. It consists of an index display, a data object window, and an optional label. It can accept various data types.
- ASCII** American Standard Code for Information Interchange.
- asynchronous execution** Mode in which multiple processes share processor time. For example, one process executes while others wait for interrupts during device I/O or while waiting for a clock tick.

auto-indexing Capability of loop structures to disassemble and assemble arrays at their borders. As an array enters a loop with auto-indexing enabled, the loop automatically disassembles it with scalars extracted from one-dimensional arrays, one-dimensional arrays extracted from two-dimensional arrays, and so on. Loops assemble data into arrays as they exit the loop according to the reverse of the same procedure.

autoscaling Ability of scales to adjust to the range of plotted values. On graph scales, this feature determines maximum and minimum scale values, as well.

autosizing Automatic resizing of labels to accommodate text that you enter.

B

block diagram Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the block diagram window of the VI.

Boolean controls and indicators Front panel objects used to manipulate and display or input and output Boolean (TRUE or FALSE) data. Several styles are available, such as switches, buttons and LEDs.

breakpoint A pause in execution.

Breakpoint tool Tool used to set a breakpoint on a VI, node or wire.

broken VI VI that cannot be compiled or run; signified by a broken arrow in the run button.

Bundle node Function that creates clusters from various types of elements.

byte stream file File that stores data as a sequence of ASCII characters or bytes.

C

case One subdiagram of a Case Structure.

Case Structure Conditional branching control structure, which executes one and only one of its subdiagrams based on its input. It is the combination of the IF, THEN, ELSE, and CASE statements in control flow languages.

chart *See* scope chart, strip chart, and sweep chart.

CIN	<i>See</i> Code Interface Node.
cloning	To make a copy of a control or some other LabVIEW object by clicking the mouse button while pressing the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key and dragging the copy to its new location. (Sun and HP-UX) You can also clone an object by clicking on the object with the middle mouse button and then dragging the copy to its new location.
cluster	A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
cluster shell	Front panel object that contains the elements of a cluster.
Code Interface Node	Special block diagram node through which you can link conventional, text-based code to a VI.
coercion	The automatic conversion LabVIEW performs to change the numeric representation of a data element.
coercion dot	Glyph on a node or terminal indicating that the numeric representation of the data element changes at that point.
Color tool	Tool used to color objects and backgrounds.
Color Copy tool	Tool used to copy colors for pasting with the Color tool.
compile	Process that converts high-level code to machine-executable code. LabVIEW automatically compiles VIs before they run for the first time after creation or alteration.
conditional terminal	The terminal of a While Loop that contains a Boolean value that determines whether the VI performs another iteration.
connector	Part of the VI or function node that contains its input and output terminals, through which data passes to and from the node.
connector pane	Region in the upper right corner of a front panel that displays the VI terminal pattern. It underlies the icon pane.
constant	<i>See</i> universal and user-defined constants.
continuous run	Execution mode in which a VI is run repeatedly until the operator stops it. You enable it by clicking on the continuous run button.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically.

control flow	Programming system in which the sequential order of instructions determines execution order. Most conventional text-based programming languages, such as C, Pascal, and BASIC, are control flow languages.
Controls palette	Palette containing front panel controls and indicators.
conversion	Changing the type of a data element.
count terminal	The terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.
CPU	Central Processing Unit.
current VI	VI whose front panel, block diagram, or icon editor window is the active window.
custom PICT controls and indicators	Controls and indicators whose parts can be replaced by graphics you supply.
D	
data acquisition	Process of acquiring data, typically from A/D or digital input plug-in boards.
data dependency	Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. <i>See also</i> artificial data dependency.
data flow	Programming system consisting of executable nodes in which nodes execute only when they have received all required input data and produce output automatically when they have executed. LabVIEW is a dataflow system.
data logging	Generally, to acquire data and simultaneously store it in a disk file. LabVIEW file I/O functions can log data.
data storage formats	The arrangement and representation of data stored in memory.
data type descriptor	Code that identifies data types, used in data storage and representation.
datalog file	File that stores data as a sequence of records of a single, arbitrary data type that you specify when you create the file. While all the records in a datalog file must be of a single type, that type can be complex; for instance, you can specify that each record is a cluster containing a string, a number, and an array.
Description box	Online documentation for a LabVIEW object.

destination terminal	<i>See</i> sink terminal.
dialog box	An interactive screen with prompts in which you specify additional information needed to complete a command.
dimension	Size and structure attribute of an array.
drag	To drag the mouse cursor on the screen to select, move, copy, or delete objects.

E

empty array	Array that has zero elements, but has a defined data type. For example, an array that has a numeric control in its data display window but has no defined values for any element is an empty numeric array.
EOF	End-of-File. Character offset of the end of file relative to the beginning of the file (that is, the EOF is the size of the file).
execution highlighting	Feature that animates VI execution to illustrate the data flow in the VI.
external routine	<i>See</i> shared external routine.

F

file refnum	An identifier that LabVIEW associates with a file when you open it. You use the file refnum to specify that you want a function or VI to perform an operation on the open file.
flattened data	Data of any type that has been converted to a string, usually, for writing it to a file.
For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to conventional code: For $i=0$ to $n-1$, do
Formula Node	Node that executes formulas that you enter as text. Especially useful for lengthy formulas that would be cumbersome to build in block diagram form.
frame	Subdiagram of a Sequence Structure.
free label	Label on the front panel or block diagram that does not belong to any other object.

front panel	The interactive user interface of a VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.
function	Built-in execution element, comparable to an operator, function, or statement in a conventional language.
Functions palette	Palette containing block diagram structures, constants, communication features, and VIs.

G

G	The LabVIEW graphical programming language.
global variable	Non-reentrant subVI with local memory that uses an uninitialized shift register to store data from one execution to the next. The memory of copies of these subVIs is shared and thus can be used to pass global data between them.
glyph	A small picture or icon.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.
graph control	Front panel object that displays data in a Cartesian plane.

H

handle	Pointer to a pointer to a block of memory; handles reference arrays and strings. An array of strings is a handle to a block of memory containing handles to strings.
Help window	Special window that displays the names and locations of the terminals for a function or subVI, the description of controls and indicators, the values of universal constants, and descriptions and data types of control attributes. The window also accesses the online reference.
hex	Hexadecimal. A base-16 number system.
hierarchical palette	Palette that contains palettes and subpalettes.

Hierarchy window	Window that graphically displays the hierarchy of VIs and subVIs.
housing	Nonmoving part of front panel controls and indicators that contains sliders and scales.
I	
icon	Graphical representation of a node on a block diagram.
Icon Editor	Interface similar to that of a paint program for creating VI icons.
icon pane	Region in the upper right corner of the front panel and block diagram that displays the VI icon.
IEEE	Institute for Electrical and Electronic Engineers.
indicator	Front panel object that displays output.
Inf	Digital display value for a floating-point representation of infinity.
inplace execution	Ability of a function or VI to reuse memory instead of allocating more.
instrument driver	VI that controls a programmable instrument.
I/O	Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.
iteration terminal	The terminal of a For Loop or While Loop that contains the current number of completed iterations.
L	
label	Text object used to name or describe other objects or regions on the front panel or block diagram.
Labeling tool	Tool used to create labels and enter text into text windows.
LabVIEW	Laboratory Virtual Instrument Engineering Workbench.
LED	Light-emitting diode.
legend	Object owned by a chart or graph that display the names and plot styles of plots on that chart or graph.

M

marquee	A moving, dashed border that surrounds selected objects.
matrix	Two-dimensional array.
MB	Megabytes of memory.
menu bar	Horizontal bar that contains names of main menus.

N

NaN	Digital display value for a floating-point representation of <i>not a number</i> , typically the result of an undefined operation, such as $\log(-1)$.
nodes	Execution elements of a block diagram consisting of functions, structures, and subVIs.
nondisplayable characters	ASCII characters that cannot be displayed, such as newline, tab, and so on.
not-a-path	A predefined value for the path control that means the path is invalid.
not-a-refnum	A predefined value that means the refnum is invalid.
numeric controls and indicators	Front panel objects used to manipulate and display or input and output numeric data.

O

object	Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures.
Object pop-up menu tool	Tool used to access an object's pop-up menu.
Operating tool	Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

P

palette	Menu of pictures that represent possible options.
platform	Computer and operating system.
plot	A graphical representation of an array of data shown either on a graph or a chart.
polymorphism	Ability of a node to automatically adjust to data of different representation, type, or structure.
pop up	To call up a special menu by clicking (usually on an object) with the right mouse button (on Window, Sun, and HP-UX) or while holding down the command key (on the Macintosh).
pop-up menus	Menus accessed by command-clicking, usually on an object. Menu options pertain to that object specifically.
Positioning tool	Tool used to move, select, and resize objects.
probe	Debugging feature for checking intermediate values in a VI.
Probe tool	Tool used to create probes on wires.
programmatic printing	Automatic printing of a VI front panel after execution.
pull-down menus	Menus accessed from a menu bar. Pull-down menu options are usually general in nature.

R

reentrant execution	Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.
representation	Subtype of the numeric data type, of which there are signed and unsigned byte, word, and long integers, as well as single-, double-, and extended-precision floating-point numbers, both real and complex.
resizing handles	Angled handles on the corner of objects that indicate resizing points.
ring control	Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

S

scalar	Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans, strings, and clusters are explicitly singular instances of their respective data types.
scale	Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
scope chart	Numeric indicator modeled on the operation of an oscilloscope.
Scroll tool	Tool used to scroll windows.
sequence local	Terminal that passes data between the frames of a Sequence Structure.
Sequence Structure	Program control structure that executes its subdiagrams in numeric order. Commonly used to force nodes that are not data-dependent to execute in a desired order.
shared external routine	Subroutine that can be shared by several CIN code resources.
shift register	Optional mechanism in loop structures used to pass the value of a variable from one iteration of a loop to a subsequent iteration.
sink terminal	Terminal that absorbs data. Also called a destination terminal.
slider	Moveable part of slide controls and indicators.
source terminal	Terminal that emits data.
string controls and indicators	Front panel objects used to manipulate and display or input and output text.
strip chart	A numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.
structure	Program control element, such as a Sequence, Case, For Loop, or While Loop.
stub VI	A nonfunctional prototype of a subVI that is created by the user. It has inputs and outputs, but is incomplete. It is used during early planning stages of VI design as a place holder for future VI development.
subdiagram	Block diagram within the border of a structure.

subVI	VI used in the block diagram of another VI; comparable to a subroutine.
sweep chart	Similar to scope chart; except a line sweeps across the display to separate old data from new data.

T

table-driven execution	A method of execution in which individual tasks are separate cases in a Case Structure that is embedded in a While Loop. are specified Sequences as arrays of case numbers.
terminal	Object or region on a node through which data passes.
tool	Special LabVIEW cursor you can use to perform specific operations.
toolbar	Bar containing command buttons that you can use to run and debug VIs.
Tools palette	Palette containing tools you can use to edit and debug front panel and block diagram objects.
top-level VI	VI at the top of the VI hierarchy. This term distinguishes the VI from its subVIs.
tunnel	Data entry or exit terminal on a structure.
type descriptor	<i>See</i> data type descriptor.

U

universal constant	Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, for example, pi.
user-defined constant	Block diagram object that emits a value you set.

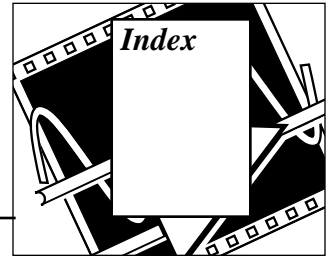
V

V	Volts.
VI	<i>See</i> virtual instrument.

VI library	Special file that contains a collection of related VIs for a specific use.
virtual instrument	LabVIEW program; so called because it models the appearance and function of a physical instrument.

W

While Loop	Loop structure that repeats a section of code until a condition is met. Comparable to a Do loop or a Repeat-Until loop in conventional programming languages.
wire	Data path between nodes.
Wiring tool	Tool used to define data paths between source and sink terminals.



A

- Acquisition switch for stopping VIs, 1-8
- Add Element option, 3-15
- Add function
 - polymorphism examples, 4-8 to 4-9
 - shift register example, 3-17
- Add Input option, 5-11
- Add Output option, 5-11
- Add Shift Register option, 3-14
- Alignment ring, Vertical Centers axis, 1-17
- Analysis palette, 4-14
- analysis VIs
 - array example, 4-12 to 4-14
 - examples in analysis directory, 4-12
- appending data to file, 6-14 to 6-16
 - block diagram, 6-15 to 6-16
 - front panel, 6-14
- Apply Changes option, File Menu, 7-10
- array controls, 4-1
- Array Max & Min function, 4-14
- array shell
 - creating array controls and indicators, 4-1
 - placing in array, 4-2
- Array Size function, 4-18
- Array Subset function, 4-18 to 4-19
- arrays, 4-1 to 4-25
 - Array Subset function, 4-18 to 4-19
 - auto-indexing
 - Initialize Array function, 4-11 to 4-12
 - input arrays, 4-9 to 4-12
 - setting For Loop count, 4-10 to 4-11
 - Build Array function, 4-16 to 4-18
 - creating and initializing, 4-15
 - using Build Array function, 4-16 to 4-18
 - creating with auto-indexing, 4-2 to 4-9
 - block diagram, 4-4 to 4-7
 - front panel, 4-2 to 4-3
 - multiplot waveform graphs, 4-7 to 4-8
 - customizing graphs, 4-23
 - data acquisition arrays, 4-25
 - definition, 4-1
 - efficient memory usage, 4-23
 - finding size of, 4-18
 - graph and analysis VIs, 4-12 to 4-14
 - block diagram, 4-13 to 4-14
 - front panel, 4-13
 - graph examples, 4-25
 - Index Array function, 4-19 to 4-22
 - indexes
 - one-dimensional (illustration), 4-1
 - range for, 4-1
 - Initialize Array function, 4-11 to 4-12
 - initializing, 4-15
 - intensity plots, 4-25
 - one-dimensional (illustration), 4-1
 - polymorphism, 4-8 to 4-9
 - single-precision vs. double precision, 4-23
 - slicing off dimensions, 4-20 to 4-22
 - types allowed, 4-1
- arrow keys for nudging objects, 9-2
- artificial data dependency, 5-17
- ASCII byte stream file format, 6-9
- ASCII waveforms, 8-21
- Attribute Node, 11-2
- auto-indexing
 - array processing, 4-9 to 4-12
 - creating array with auto-indexing, 4-2 to 4-9

- definition, 4-4
- enabling and disabling (note), 4-11
- setting For Loop count, 4-10 to 4-11

Autoscale Y submenu, 4-3

autoscaling of graph input

- default action, 4-3
- disabling, 4-3

axes

- customizing Y axis, 3-20
- modifying text format (note), 3-22
- rescaling, 3-21

B

bad wires. *See also* wiring block diagrams.

binary byte stream file format

- advantages and disadvantages, 6-22
- definition, 6-9
- example in strings.llb, 6-22

binary waveforms, 8-22

block diagram, 1-9 to 1-11

- analogous to programs, 1-10
- avoiding oversized diagrams, 10-5
- building, 1-24 to 1-31
 - activating Help window, 1-26
 - bad wires, 1-30
 - broken wires (note), 2-8
 - debugging techniques, 2-9 to 2-12
 - deleting wires, 1-30
 - Divide function, 2-8
 - Multiply function, 1-24, 2-8
 - numeric constant, 1-24, 2-8
 - selecting wires, 1-29
 - showing terminals, 1-28
 - string constant, 1-25
 - Subtract function, 2-8
 - subVIs, 2-7 to 2-12
 - wire stretching, 1-29
 - wiring techniques, 1-27
- component parts, 1-9 to 1-11
- definition, 1-3
- opening, 1-10
- subVI diagram (illustration), 1-13

block diagram examples

- appending data to file, 6-15 to 6-16
- array created with auto-indexing, 4-4 to 4-9
- Case structure, 5-2 to 5-4
- converting and concatenating strings, 6-3 to 6-6
- debugging VIs, 9-7 to 9-9
- For Loop, 3-13 to 3-14
- Formula node, 5-15 to 5-16
- frequency response test VI, 8-15 to 8-17
- graph and analysis VI, 4-13 to 4-14
- reading data from file, 6-17 to 6-18
- Sequence structure, 5-8 to 5-10
- serial port communication, 8-6 to 8-7
- shift register, 3-17 to 3-23
- subsetting strings, 6-8 to 6-9
- subVI Node Setup options, 7-3 to 7-9
- test sequencer VI, 8-18 to 8-19
- True and False cases, 4-9 to 4-10
- While Loop, 3-3 to 3-9
 - writing to spreadsheet file, 6-12 to 6-13

block diagram window, nudging objects

- in, 9-2

Boolean Case structure, 5-2 to 5-3

Boolean constant

- appending data to file example, 6-16
- writing to spreadsheet file example, 6-13

Boolean controls and indicators

- mechanical actions, 3-6 to 3-7
- purpose and use, 1-20

Boolean palette, 3-2

broken run button, 9-5

Build Array function

- adding and removing inputs, 4-16
- array example, 4-8
- creating and initializing arrays, 4-16
- Formula node example, 5-16
- illustration, 4-16
- purpose and use, 4-16 to 4-18

building VIs. *See also* VIs.

Bundle function

- array created with auto-indexing, 4-4
- graph and analysis VI example, 4-14

- order of inputs (note), 3-19
- resizing icon, 4-4
- shift register example, 3-19
- button switches, aligning, 1-17
- Bytes at Serial Port VI, 8-4, 8-7, 8-20

C

- Case structure, 5-1 to 5-4
 - block diagram, 5-2 to 5-4
 - Boolean by default, 5-2 to 5-3
 - defining output tunnel for each case (note), 5-4
 - front panel, 5-1
 - location inside While Loop, 1-11
 - purpose and use, 1-11
 - testing While Loops before execution, 3-24 to 3-25
 - VI logic, 5-4
- Change to Array option, 4-17
- Change to Indicator option, 9-4
- chart modes
 - illustration, 3-22
 - scope chart, 3-23
 - strip chart, 3-23
 - sweep chart, 3-23
- charts. *See also* graphs.
 - checking data types for inclusion, 4-5
 - clearing chart in While Loop, 3-5
 - colors, 3-21
 - compared with graphs, 4-2
 - customizing, 3-20 to 3-22
 - effect of chart size on display of axis scales (note), 3-21
 - examples, 3-1
 - faster chart updates, 3-24
 - intensity plots, 4-25
 - legends, 3-21
 - modifying axis text format (note), 3-22
 - modifying while running, 3-21
 - multiplot charts, 3-19 to 3-20
 - order of plots determined by Bundle function inputs, 3-19
 - overlaid plots, 3-20
 - plot line style, 3-21
 - point style, 3-21
 - shift register example, 3-16 to 3-22
 - stacked vs. overlaid plots, 3-24
 - temperature waveform chart, 4-13
 - waveform chart used with While Loop, 3-1 to 3-5
 - Y axis, customizing, 3-20
- Clear Chart option, Data Operations pop-up menu, 3-5
- Close option
 - File menu, 1-13
- Cluster palette, 3-19
- clusters
 - analogous to Pascal records and C structs, 4-4
 - array created with auto-indexing, 4-4 to 4-5
 - definition, 4-4
- coercion dot, 3-11
- Color tool, 1-17, 3-2
- colors
 - charts, 3-21
 - picking color from object, 9-4
 - round LED, 1-17
 - transparent border for free label, 3-2
- Comparison palette, 5-3
- conditional terminal, 3-4
- connectors. *See also* icons; terminals.
 - connecting subVIs to block diagrams, 1-12 to 1-13
 - creating, 2-4 to 2-5
 - programming considerations, 10-3 to 10-5
 - specifying inputs and outputs to VI (note), 2-5
 - subVI Node Setup options example, 7-5
- continuous run button
 - running VIs, 1-31
 - using looping structure instead of (note), 1-31
- Control Editor
 - invoking, 7-10

- purpose and use, 11-3
 - saving custom control as type definition
 - or strict type definition, 7-14
 - controls
 - automatic creation of terminal, 2-7
 - Boolean controls and indicators, 1-20
 - configuring using pop-up menus, 1-20
 - fixing incorrectly wired controls, 9-4
 - numeric controls and indicators, 1-19
 - simulating control/indicator, 7-13 to 7-14
 - using as inputs (note), 2-5
 - Controls palette
 - Numeric palette, 1-23
 - Select a Control Option, 7-10
 - String & Table palette, 6-1
 - count terminal, For Loop, 3-10
 - Ctrl key equivalents for menu options, 9-1
 - custom controls
 - importing pictures, 7-10 to 7-12
 - invoking Control Editor, 7-10
 - saving, 7-10
 - saving as type definition or strict type
 - definition, 7-14
 - customer communication, *xxi*, A-1
 - customer education, *xxi*
 - customizing VIs. *See also* VI Setup options.
- D**
- data acquisition arrays, 4-25
 - data acquisition boards
 - available data acquisition VIs, 8-2
 - features, 8-2
 - platforms supported, 8-2
 - data dependency
 - artificial, 5-17
 - missing, in program structure, 10-9
 - data flow programming
 - artificial data dependency, 5-17
 - controlling execution with Sequence
 - structure, 5-9
 - debugging with execution highlighting,
 - 2-10 to 2-11
 - missing dependencies, 10-9
 - order of subVI node execution, 2-11
 - Data Range option, 5-7
 - data range, setting, 5-7
 - data types, checking for inclusion in
 - graphs, 4-5
 - datalog file format, 6-21
 - advantages, 6-21
 - definition, 6-9, 6-21
 - debugging VIs
 - development tips, 9-1 to 9-5
 - example
 - block diagram, 9-7 to 9-9
 - front panel, 9-6
 - execution highlighting, 9-6
 - subVI example, 2-10 to 2-11
 - finding errors, 9-5
 - opening front panels of subVIs, 9-9
 - to 9-10
 - single stepping through VIs, 9-5
 - subVI example, 2-9 to 2-12
 - decrementing faster, 9-3
 - deleting wires, 1-29, 9-5
 - Demo Fluke 8840A subVI, 8-15
 - Demo Tek FG 5010 subVI, 8-15
 - Demo Voltage Read VI, 1-21, 1-24, 1-26
 - Description option, 1-32 to 1-34
 - digital control
 - illustration, 1-19
 - label for, 1-15
 - repositioning, 1-15
 - Digital Display option, Show pop-up
 - menu, 3-2
 - digital indicator
 - creating owned label, 1-16
 - illustration, 1-17
 - Digital Thermometer VI, 4-14, 7-9
 - Disable Indexing option, 4-20
 - Distribution ring, 1-17
 - Divide function
 - adding to block diagram, 2-8
 - Sequence structure example, 5-10
 - shift register example, 3-18
 - documentation

- advanced topics, 11-2 to 11-4
- conventions used in manual, *xix* to *xx*
- organization of manual, *xvii* to *xix*
- other useful documentation, 11-1 to 11-2
- related documentation, *xx*
- documenting VIs, 1-32 to 1-34
 - using Show VI Info option, 1-32
 - viewing object descriptions, 1-32 to 1-34
- duplicating objects, 9-3

E

- Edit Control option, Edit menu, 7-11
- Edit Icon option, 2-2
- Edit menu
 - Edit Control option, 7-11
 - Remove Bad Wires option, 1-30
- editing VIs, 1-14 to 1-18
- efficient memory usage with arrays, 4-23
- Empty Path constant, 6-15
- Enable Indexing option, 4-20
- enter button, 1-8
- error handler VIs
 - Find First Error, 8-20
 - General Error Handler, 8-20
 - Simple Error Handler, 8-7, 8-20
- error handling, 10-7 to 10-8
- example files
 - examples directory, 1-4
 - where to find, 11-1
- execution highlighting for debugging VIs
 - buttons for, 9-6
 - subVI example, 2-10 to 2-11
 - VI example, 9-7 to 9-9
- Execution Options
 - subVI Node Setup options example, 7-5 to 7-6
- Extract Numbers VI
 - converting ASCII waveforms, 8-21
 - reading data from file example, 6-18

F

- file I/O

- appending data to file, 6-14 to 6-16
 - block diagram, 6-15 to 6-16
 - front panel, 6-14
- ASCII byte stream format, 6-9
- avoiding writing data to VI libraries (caution), 6-13
- binary byte stream format, 6-9, 6-22
- datalog format, 6-9, 6-21
- examples in `smplefile.llb`, 6-20
- file utility functions, 6-10
- paths, 6-19
- reading data from file, 6-16 to 6-18
 - block diagram, 6-17 to 6-18
 - front panel, 6-17
- refnums, 6-19
 - specifying files, 6-18 to 6-19
- writing to spreadsheet file, 6-11 to 6-13
 - block diagram, 6-12 to 6-13
 - front panel, 6-12
- file I/O functions
 - Read Characters From File VI, 6-10
 - Read From Spreadsheet File VI, 6-10
 - Read Lines From File VI, 6-10
 - Write Characters To File VI, 6-10
 - Write To Spreadsheet File VI, 6-10
- File I/O palette, 6-9
- File menu
 - Apply Changes option, 7-10
 - Close option, 1-13
 - Save option, 1-34
- file utility functions
 - Read Characters From File VI, 6-18
- files for LabVIEW, 1-4
- Find First Error VI, 8-20
- Flip Horizontal option, 7-5
- floating-point numbers
 - double-precision, as default representation, 3-10
 - rounding (note), 3-11
- flow of data. *See also* data flow programming.
- folders for VIs stored in VI libraries (illustration), 1-35

- fonts for labels, changing size of, 1-16
- For Loop, 3-9 to 3-14
 - auto-indexing
 - array processing, 4-10 to 4-12
 - creating array with auto-indexing, 4-2 to 4-9
 - definition, 4-4
 - setting count for For Loop, 4-10 to 4-11
 - avoiding continuous updating of indicators (note), 3-14
 - block diagram, 3-13 to 3-14
 - count terminal, 3-10
 - equivalent pseudocode, 3-10
 - frequency response test VI, 8-15 to 8-17
 - front panel, 3-12
 - iteration terminal, 3-10
 - location inside While Loop, 1-11
 - numeric conversion, 3-10 to 3-11
 - placing on block diagram, 3-9
 - programming considerations, 10-6 to 10-7
- Format & Precision option, 5-5
- Format Into String function
 - appending data to file example, 6-15
 - string concatenation example, 6-3
- Formula node, 5-11 to 5-16
 - block diagram, 5-15 to 5-16
 - conditional assignment (example), 5-12
 - creating input and output terminals, 5-11
 - definition, 5-11
 - frequency response test VI, 8-15 to 8-17
 - front panel, 5-14
 - Help window for displaying operators and functions, 5-12
 - illustration, 5-11, 5-15
 - purpose and use, 5-11
 - semicolon (;) terminating formula statements, 5-11, 5-15
 - variable names (note), 5-15
- free labels
 - changing font size, 1-16
 - duplicating, 1-16
- frequency response test VI, 8-13 to 8-17. *See also* test sequencer VI.
 - block diagram, 8-15 to 8-17
 - front panel, 8-14
- From Exponential/Fract/Eng function, 6-8
- front panel, 1-5 to 1-9. *See also* front panel examples.
 - building
 - subVIs, 2-6 to 2-7
 - VIs, 1-23
 - definition, 1-3
 - opening, 1-6 to 1-7
 - toolbar, 1-7 to 1-8
- front panel examples
 - appending data to file, 6-14
 - array created with auto-indexing, 4-2 to 4-3
 - Case structure, 5-1
 - converting and concatenating strings, 6-2 to 6-3
 - debugging VIs, 9-6
 - For Loop, 3-12
 - Formula Node, 5-14
 - frequency response test VI, 8-14
 - graph and analysis VI, 4-13
 - reading data from file, 6-17
 - Sequence structure, 5-5 to 5-7
 - serial port communication, 8-5
 - subsetting strings, 6-7
 - subVI Node Setup options, 7-4
 - test sequencer VI, 8-17
 - While Loop, 3-2 to 3-3
 - writing to spreadsheet file, 6-12
- Front Panel toolbar, 1-7 to 1-8
- front panel window, nudging objects in, 9-2
- Functions palette
 - File I/O palette, 6-10
 - Instrument I/O palette, 8-3
 - Select a VI option, 2-6, 2-7
 - String palette, 6-2
- functions. *See also* specific functions.

G

G programming language, 1-3

General Error Handler VI, 8-20
 General Purpose Interface Bus. *See also* GPIB.
 Generate Waveform VI, 4-2 to 4-3
 Get Date/Time String VI, 7-8
 Get Operator Info VI, 7-8
 global variables
 disadvantages, 10-3
 purpose and use, 11-3
 GPIB
 definition, 8-3
 examples of functions, 8-4
 using VISA rather than (note), 8-4
 GPIB Read function, 8-19
 GPIB Status function, 8-19
 GPIB Write function, 8-19
 graph cursors, 4-24
 graph indicators, 4-2
 graph VIs
 array example, 4-12 to 4-14
 graphical programming language (G), 1-3
 graphs. *See also* charts.
 autoscaling of input, 4-3
 checking data types for inclusion, 4-5
 compared with charts, 4-2
 customizing, 4-23
 examples in graphs directory, 4-2, 4-25
 intensity plots, 4-25
 multiplot graphs, 4-7 to 4-8
 showing or hiding optional parts, 4-23
 types of graphs, 4-2
 Greater Or Equal to 0? function, 5-3

H

Help menu, Show Help option, 1-26
 Help window
 activating, 1-26
 Formula node operators and functions,
 displaying, 5-12
 online help for subVI nodes, 2-15
 Hewlett Packard 34401A Multimeter
 instrument driver example
 block diagram, 8-10 to 8-13
 front panel, 8-9

hierarchy of VIs
 definition, 1-3
 description, 1-11, 2-1
 illustration, 1-12
 programming considerations, 10-2
 to 10-3
 Horizontal Centers distribution
 Distribution ring, 1-17
 horizontal motion, limiting objects to, 9-3
 hot spot of Wiring tool, 1-27
 HP34401A Config Measurement VI, 8-10
 HP34401A Config Trigger VI, 8-10
 HP34401A Read Measurement VI, 8-10

I

Icon Editor
 buttons, 2-3
 invoking, 2-2
 tools, 2-2 to 2-3
 icons. *See also* connectors.
 creating, 2-2 to 2-3
 grouping into lower level VI, 2-1
 representing VIs in block diagram of
 other VIs, 1-12
 subVI Node Setup options example, 7-5
 IEEE 488. *See also* GPIB.
 Import Picture option, 7-11
 Increment function, 5-10
 incrementing faster, 9-3
 Index Array function
 illustration, 4-19
 purpose and use, 4-19 to 4-22
 rules governing slicing of arrays, 4-21
 to 4-22
 slicing dimensions from
 multi-dimensional arrays, 4-20 to 4-22
 indexes for arrays, 4-20. *See also*
 auto-indexing.
 disabling and enabling
 one-dimensional (illustration), 4-1
 range for, 4-1
 indicators
 automatic creation of terminal, 2-7

- avoiding continuous updating in For Loop (note), 3-14
 - Boolean controls and indicators, 1-20
 - configuring, 1-20
 - fixing incorrectly wired controls, 9-4
 - numeric controls and indicators, 1-19
 - simulating control/indicator, 7-13 to 7-14
 - using as outputs (note), 2-5
 - Initialize Array function, 4-11 to 4-12
 - installation of LabVIEW, 1-4
 - instrument drivers, 8-8 to 8-13
 - available library of drivers, 8-8
 - Hewlett Packard 34401A Multimeter example, 8-9 to 8-13
 - block diagram, 8-10 to 8-13
 - front panel, 8-9
 - purpose and use, 8-8
 - using as subVIs, 8-8
 - Instrument I/O palette
 - GPIB functions, 8-3
 - Serial palette, 8-4
 - intensity graphs, 4-2, 4-25
 - iteration terminal
 - definition, 3-4
 - For Loop, 3-10
 - Formula node example, 5-16
- K**
- knob control, adding to front panel for While Loop, 3-3
- L**
- Labeling tool
 - changing font size, 1-16
 - creating free label, 3-2
 - entering or changing text in string controls, 6-1
 - labels
 - changing font size, 1-16
 - clicking outside text box (note), 1-23
 - creating owned label for digital indicator, 1-16
 - duplicating free labels, 1-16
 - front panel objects, 1-23
 - owned labels, 1-15
 - repositioning, 1-15 to 1-16
 - scale for knob, 3-3
 - vertical switch (example), 3-2
 - LabVIEW
 - files, 1-4
 - how LabVIEW works, 1-3
 - installation, 1-4
 - overview, 1-2
 - training courses, 1-1
 - Latch Until Released action, 3-7
 - Latch When Pressed action, 3-7
 - Latch When Released action, 3-7
 - left terminal, shift registers
 - accessing values from previous iterations, 3-15
 - purpose, 3-14
 - Legend option, Show pop-up menu, 5-14
 - legends for charts
 - creating for graph, 5-14
 - positioning and modifying, 3-21
 - libraries. *See also* VI libraries.
 - list controls, when to use, 11-3
 - local variables
 - acting as multiple terminal to front panel control or indicator, 7-14
 - disadvantages, 11-2 to 11-3
 - loops. *See also* For Loop; While Loop.
- M**
- manual. *See also* documentation.
 - Match Pattern function, 8-12
 - Max & Min function, 3-14
 - Mean VI, 4-14
 - mechanical actions of Boolean controls, 3-6 to 3-7
 - memory, efficient usage with arrays, 4-23
 - menus for LabVIEW
 - Ctrl key equivalents for menu options, 9-1
 - rotating through tools in Tools palette,

9-2
 modular programming. *See also* program design.
 mouse button for activating pop-up menus, 1-13, 1-23
 multi-dimensional arrays, slicing dimensions from, 4-20 to 4-22
 multiplot graphs
 array example, 4-7 to 4-8
 While Loop example, 3-19 to 3-20
 Multiply function
 adding to block diagram, 1-24, 2-8
 polymorphism, 4-8
 Sequence structure example, 5-9
 While Loop, 3-9

N

nodes, replacing, 9-4
 Not Equal? function, 5-10
 Not function, 7-4
 numeric constants
 adding to block diagram, 1-24, 2-8
 array created with auto-indexing, 4-5 to 4-6
 Case structure example, 5-3
 For Loop example, 3-13
 Formula node example, 5-15
 shift register example, 3-18
 While Loop example, 3-9
 numeric controls and indicators
 default representation, 3-10
 modifying numeric format, 5-5
 numeric conversion, 3-10 to 3-11
 purpose and use, 1-19
 numeric conversion, 3-10 to 3-11
 Numeric palette, 2-8
 Functions palette, 1-24, 2-8
 Numeric palette, Controls palette, 1-23

O

objects
 limiting to horizontal or vertical

 motion, 9-3
 nudging with arrow keys, 9-2
 picking color from, 9-4
 One Button Dialog function, 5-3
 Operating tool
 entering or changing text in string controls, 6-1
 manipulating slide controls, 1-8
 purpose and use, 1-8
 owned labels
 characteristics, 1-15
 creating for digital indicator, 1-16

P

Parse String VI, 6-7
 path control, 6-19
 path indicator, 6-19
 path, definition of, 6-19
 Patterns option, 7-5
 Pi constant, 4-8
 Pick Line & Append function, 8-12
 pictures, importing into custom controls, 7-10 to 7-12
 plots. *See also* charts.
 intensity plots, 4-25
 line style for charts, 3-21
 order determined by Bundle function inputs, 3-19
 overlaid plots, 3-20
 shift register example, 3-16 to 3-22
 stacked vs. overlaid plots, 3-24
 point style for charts, 3-21
 polymorphism, 4-8 to 4-9
 pop-up menus
 configuring controls and indicators, 1-20
 illustration, 1-20
 right mouse button for activating, 1-13, 1-23
 Positioning tool
 enlarging string controls, 6-1
 moving objects in front panel, 1-16
 probe
 debugging subVIs, 2-9 to 2-10

- debugging VIs (example), 9-9
- program design, 10-1 to 10-10. *See also* data flow programming; debugging VIs.
 - avoiding global variables, 10-3
 - creating stub VIs, 10-2 to 10-3
 - determining user requirements, 10-1
 - diagramming style, 10-5 to 10-10
 - avoiding overlapped diagrams, 10-5
 - avoiding Sequence structure overuse, 10-10
 - checking for errors, 10-7 to 10-8
 - left-to-right layouts, 10-7
 - missing dependencies, 10-9
 - putting common operations into loops, 10-6 to 10-7
 - studying examples, 10-10
 - hierarchy design, 10-2 to 10-3
 - modular programming, 10-3
 - hierarchical nature of VIs and, 1-3
 - planning ahead with connector patterns, 10-3 to 10-5
 - top-down design, 10-1 to 10-3
 - writing the program, 10-3
- program design., 9-1
- program design, 9-1 to 9-5. *See also* data flow programming.

R

- Random Number (0–1) function
 - For Loop example, 3-13
 - Sequence structure example, 5-9
 - shift register example, 3-17
 - While Loop example, 3-4
- range error symbol, 5-7
- Read Characters From File VI
 - purpose, 6-10
 - reading data from file example, 6-18
- Read from Datalog File VI, 6-21
- Read From Spreadsheet File VI, 6-10
- Read Lines From File VI, 6-10
- reading data from file, 6-16 to 6-18
 - block diagram, 6-17 to 6-18
 - front panel, 6-17

- readme.vi, 11-1
- refnums, file, 6-19
- Remove Bad Wires option, Edit menu, 1-30
- Remove Dimension option, 4-12
- Replace option, 9-4
- representation of numeric values
 - default representation, 3-10
 - modifying numeric format, 5-5
 - numeric conversion, 3-10 to 3-11
- resizing of round LED, 1-17
- right mouse button for activating pop-up menus, 1-13, 1-23
- right terminal, shift registers, 3-14
- ring controls
 - adding items to, 9-3
 - when to use, 11-3
- rotating through tools in Tools palette, 9-2
- round LED
 - changing color, 1-17
 - illustration, 1-17
 - resizing, 1-17
- Round to Nearest function, 5-10
- rounding to nearest integer, 3-11
- run button, 1-31

S

- Save option, File menu, 1-34
- saving VIs
 - libraries for storing VIs, 1-34 to 1-35
 - procedure for, 1-34 to 1-35
- scope chart mode, 3-23
- Scrollbar option, Show pop-up menu, 3-5
- scrollbars
 - adding to While Loop, 3-5
 - minimizing space required for string controls, 6-2
- Select & Append function, 8-12
- Select a Control option, Controls palette, 7-10
- Select a VI option, Functions palette, 2-6, 2-7
- semicolon (;) terminating formula statements, 5-11
- Separate Array Values VI, 4-9, 9-6

- sequence local variable
 - creating, 5-9
 - illustration, 5-9
- Sequence structure, 5-5 to 5-10
 - block diagram, 5-8 to 5-10
 - controlling execution order of nodes, 5-9
 - front panel, 5-5 to 5-7
 - illustration, 5-9
 - modifying numeric format, 5-5
 - programming considerations, 10-10
 - setting data range, 5-7
 - subVI Node Setup options example, 7-8 to 7-9
 - timing, 5-17
- Serial palette, 8-4
- serial port VIs
 - available for serial communication, 8-4
 - Bytes at Serial Port VI, 8-4, 8-7, 8-20
 - Serial Port Init VI, 8-5, 8-6, 8-20
 - Serial Port Read VI, 8-5, 8-7
 - Serial Port Write VI, 8-5, 8-7
- serial ports, 8-4 to 8-7
 - communication example, 8-5 to 8-7
 - block diagram, 8-6 to 8-7
 - front panel, 8-5
 - serial communication, 8-4
- shift registers, 3-14 to 3-23
 - adaptation to data type of first object, 3-15
 - adding to For Loop, 3-13
 - block diagram, 3-17 to 3-23
 - customizing charts, 3-20 to 3-22
 - different chart modes, 3-22 to 3-23
 - multiplot charts, 3-19 to 3-20
 - creating, 3-14 to 3-15
 - definition, 3-14
 - front panel, 3-16 to 3-17
 - initializing
 - avoiding incorporation of old data (note), 3-18
 - For Loop example, 3-13
 - Initialize Array function, 4-11 to 4-12
 - left terminal, 3-14
 - location on While Loop, 1-10
 - remembering values from previous iterations, 3-15
 - right terminal, 3-14
 - uninitialized shift registers, uses for, 3-26 to 3-27
- Show Connector option, 1-12, 2-4
- Show Diagram option, Windows menu, 1-10, 1-13, 1-24
- Show Help option, Help Menu, 1-26
- Show Icon option, 1-13
- Show Terminals option, 1-28
- Show VI Info option for documenting VIs, 1-32
- Simple Error Handler VI, 8-20
 - error checking, 8-20
 - serial port communication example, 8-7
- Sine function, 4-8
- single stepping through VIs, 9-5, 9-8 to 9-9
- slicing dimensions from multi-dimensional arrays, 4-20 to 4-22
- slide controls, manipulating, 1-8
- slide switches, repositioning, 1-16
- spreadsheet files
 - Read From Spreadsheet File VI, 6-10
 - Write To Spreadsheet File VI, 6-10
 - writing to, 6-11 to 6-13
 - block diagram, 6-15 to 6-16
 - front panel, 6-14
- Square Root function, 5-3
- step into button, 9-5
- step out button, 9-5
- step over button, 9-5
- Stop button, 1-7, 1-9
- stopping VIs
 - Acquisition switch, 1-9
 - Stop button, 1-9
 - without interrupting I/O (note), 1-9
- strict type definition, saving custom control as, 7-14
- String & Table palette, 6-1
- string constants

- adding to block diagram, 1-25
 - appending data to file example, 6-16
 - Case structure example, 5-3
 - string controls and indicators
 - creating, 6-1
 - minimizing space, 6-2
 - string function examples
 - converting and concatenating strings, 6-2 to 6-3
 - block diagram, 6-3 to 6-6
 - front panel, 6-2 to 6-3
 - subsetting strings, 6-7 to 6-9
 - block diagram, 6-8 to 6-9
 - front panel, 6-7
 - string functions
 - Scan From String, 6-8
 - String Length, 6-3
 - String Subset, 6-8
 - String Length function, 6-3
 - String palette, 6-2
 - String Subset function, 6-8
 - String To Byte Array VI, 8-22
 - strings. *See also* string function examples.
 - creating string controls and indicators, 6-1
 - definition, 6-1
 - strip chart mode, 3-23
 - structures. *See also* Case structure; For Loop; Sequence structure; While Loop.
 - examples, 3-1
 - types of, 3-1
 - stub VIs, 10-2 to 10-3
 - Subtract function
 - adding to block diagram, 2-8
 - Sequence structure example, 5-10
 - subVI Node Setup options. *See also* VI Setup options.
 - restricted to one node only (note), 7-3
 - subVI example, 7-1 to 7-9
 - block diagram, 7-8 to 7-9
 - front panel, 7-7
 - subVI nodes
 - analogous to subroutine call, 2-6
 - online help for subVI nodes, 2-15
 - order of execution in data flow programming, 2-11
 - subVIs, 2-1 to 2-17
 - analogous to subroutines, 2-1, 2-6
 - changing, 2-12
 - creating, 2-1 to 2-5
 - block diagram, 2-7 to 2-12
 - connector, 2-4 to 2-5
 - debugging techniques, 2-9 to 2-12
 - front panel, 2-6 to 2-7
 - icon, 2-2 to 2-3
 - grouping icons into lower level VI, 2-1
 - hierarchical nature of, 2-1
 - online help for subVI nodes, 2-15
 - opening, 2-12
 - opening front panels for debugging, 9-9 to 9-10
 - operating, 2-12
 - using VIs as subVIs, 1-11, 2-6 to 2-16
 - sweep chart mode, 3-23
 - Switch Until Released action, 3-6
 - Switch When Pressed action, 3-6
 - Switch When Released action, 3-6
- ## T
- technical support, A-1
 - Temperature Status subVI, 1-12
 - Temperature System Demo VI, 1-6 to 1-9
 - terminals
 - analogous to parameters in subroutines or functions, 1-12
 - assigning to subVI, 2-4 to 2-5
 - automatic creation for controls and indicators, 2-7
 - conditional terminal, 3-4
 - connecting subVIs to other VIs, 1-12 to 1-13
 - iteration terminal, 3-4
 - programming considerations, 10-3 to 10-4
 - shift registers, 3-14 to 3-15
 - showing while wiring block

- diagrams, 1-28
- test sequencer VI., 8-17 to 8-19. *See also*
 - frequency response test VI.
 - block diagram, 8-18 to 8-19
 - front panel, 8-17
- Thermometer indicator, 2-7
- three-dimensional array, slicing, 4-21
- Tick Count (ms) function, 5-9, 5-10
- Time & Dialog palette, 3-8
- timing
 - Sequence structure, 5-17
 - While Loop, 3-7 to 3-9
- Timing.Template.vi example, 5-17
- toolbar
 - Alignment ring, 1-17
 - Distribution ring, 1-17
- Tools palette, rotating through tools in, 9-2
- top-down design. *See also* program design.
- training for LabVIEW, 1-1
- tunnels
 - defining output tunnel for each case (note), 5-4
 - wiring to output tunnel (note), 5-4
- Tutorial palette, 4-14
- tutorial.lib library, 1-4
- type definition, saving custom control as, 7-14

U

- uninitialized shift registers. *See also* shift registers.
- Update Mode submenu, 3-22
- Update Period slide control, 1-8

V

- variable names in Formula node
 - case sensitivity, 5-15
 - length considerations (note), 5-15
- Vertical Centers alignment, Alignment ring, 1-17
- vertical motion, limiting objects to, 9-3
- vertical switch

- adding to front panel, 3-2
- illustration, 1-20
- VI libraries
 - storing VIs in, 1-34 to 1-35
- VI Setup option, 7-1
- VI Setup options, 7-1 to 7-2. *See also* subVI Node Setup options.
 - Execution Options, 7-6
 - global application of (note), 7-3
 - Window Options, 7-2, 7-6
- vi.lib directory, 1-4
- virtual instruments. *See also* subVIs; VIs.
- VIs, 1-4 to 1-13. *See also* subVIs.
 - analogous to functions in programming languages, 1-3
 - block diagram
 - definition, 1-3
 - description, 1-9 to 1-11
 - building, 1-21 to 1-35
 - bad wires, 1-30
 - block diagram, 1-24 to 1-31
 - deleting wires, 1-29
 - documenting VIs, 1-32 to 1-34
 - front panel, 1-23
 - selecting wires, 1-29
 - showing terminals, 1-28
 - wire stretching, 1-29
 - wiring techniques, 1-27
 - editing, 1-14 to 1-18
 - features, 1-3
 - front panel
 - definition, 1-3
 - working with, 1-5 to 1-9
 - hierarchical structure
 - definition, 1-3
 - description, 1-11 to 1-12
 - icon/connector, 1-12 to 1-13
 - modular nature of, 1-3
 - running, 1-31
 - saving, 1-34 to 1-35
 - stopping, 1-9
- VISA
 - examples of functions, 8-3

VISA functions, 8-2

W

Wait Until Next ms Multiple function

graph and analysis VI example, 4-14

shift register example, 3-18

subVI Node Setup options example, 7-9

While Loop example, 3-9

waveform graphs

as type of graph, 4-2

Formula node, example, 5-14

temperature waveform chart, 4-13

using in array, example, 4-2 to 4-3, 4-5 to 4-6

multiplot graphs, 4-7 to 4-8

While Loop example, 3-1 to 3-5

waveform transfers, 8-21 to 8-22

ASCII waveforms, 8-21

binary waveforms, 8-22

While Loop, 3-1 to 3-9. *See also* shift registers.

adding knob control to front panel, 3-3

adding timing, 3-7 to 3-9

block diagram, 3-3 to 3-9

Case structure in, 1-11

clearing display buffer, 3-5

equivalent pseudocode, 3-4

For Loop in, 1-11

location in block diagram, 1-10

mechanical action of Boolean switches, 3-6 to 3-7

programming considerations, 10-5 to 10-7

shift registers located on, 1-10

testing before execution, 3-24 to 3-25

waveform chart used with, 3-1 to 3-5

Window Options

dialog box, 7-2

setting, 7-2

subVI Node Setup options example, 7-6

Windows menu, 1-10

Show VI Info, 1-32

wires

branches, 1-29

dashed wires vs. dotted wires (note), 1-30

junction, 1-29

nudging with arrow keys, 9-2

segments, 1-29

wiring block diagrams, 1-27 to 1-31

bad wires, 1-30

basic techniques, 1-27

bending wires, 1-27

changing direction of wire with space bar, 9-2

dashed wires vs. dotted wires (note), 1-30

deleting wires, 1-29, 9-5

selecting wires, 1-29

showing terminals, 1-28

stretching wires, 1-29

tacking wires, 1-27

Wiring tool hot spot, 1-27

Write Characters to File VI

appending data to file example, 6-16
purpose, 6-10

Write to Datalog File VI, 6-21

Write to Spreadsheet File VI

example, 6-13
purpose, 6-10

X

X button for rescaling X axis, 3-21

XY graphs, 4-2

Y

Y button for rescaling Y axis, 3-21