LabVIEW

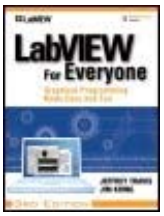
PRENTICE HALL

LabVIEW For Everyone

Graphical Programming Made Easy and Fun

JEFFREY TRAVIS
JIM KRING

THIRD EDITION



LabVIEW for Everyone: Graphical Programming Made Easy and Fun, Third Edition

By Jeffrey Travis, Jim Kring

.....

Publisher: Prentice Hall
Pub Date: July 27, 2006
Print ISBN-10: 0-13-185672-3
Print ISBN-13: 978-0-13-185672-1
Pages: 1032

Table of Contents | Index

Overview

The #1 Step-by-Step Guide to LabVIEWNow Completely Updated for LabVIEW 8!

Master LabVIEW 8 with the industry's friendliest, most intuitive tutorial: *LabVIEW for Everyone, Third Edition*. Top LabVIEW experts Jeffrey Travis and Jim Kring teach LabVIEW the easy way: through carefully explained, step-by-step examples that give you reusable code for your own projects!

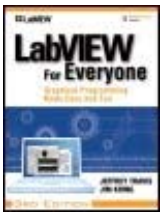
This brand-new Third Edition has been fully revamped and expanded to reflect new features and techniques introduced in LabVIEW 8. You'll find two new chapters, plus dozens of new topics, including Project Explorer, AutoTool, XML, event-driven programming, error handling, regular expressions, polymorphic VIs, timed structures, advanced reporting, and much more. Certified LabVIEW Developer (CLD) candidates will find callouts linking to key objectives on NI's newest exam, making this book a more valuable study tool than ever.

- Not just what to do: why to do it!
- Use LabVIEW to build your own virtual workbench
- Master LabVIEW's foundations: wiring, creating, editing, and debugging VIs; using controls and indicators; working with data structures; and much more
- Learn the "art" and best practices of effective LabVIEW development
- NEW: Streamline development with LabVIEW Express VIs
- NEW: Acquire data with NI-DAQmx and the LabVIEW DAQmx VIs
- NEW: Discover design patterns for error handling, control structures, state machines, queued messaging, and more
- NEW: Create sophisticated user interfaces with tree and tab controls, drag and drop,

subpanels, and more

Whatever your application, whatever your role, whether you've used LabVIEW or not, *LabVIEW for Everyone, Third Edition* is the fastest, easiest way to get the results you're after!

NEXT



LabVIEW for Everyone: Graphical Programming Made Easy and Fun, Third Edition

By Jeffrey Travis, Jim Kring

Publisher: Prentice Hall
Pub Date: July 27, 2006
Print ISBN-10: 0-13-185672-3
Print ISBN-13: 978-0-13-185672-1
Pages: 1032

Table of Contents | Index

PREV NEXT

Copyright

Library of Congress Cataloging-in-Publication Data

Travis, Jeffrey.

LabVIEW for everyone : graphical programming made easy and fun / Jeffrey Travis, James Kring. 3rd
p. cm.
ISBN 0-13-185672-3 (pbk. : alk. paper) 1. Scientific apparatus and instrumentsComputer simulation.
2. LabVIEW. I. Kring, James. II. Title.

Q183.A1T73 2006
006dc22 2006012875

Text printed in the United States on recycled paper at R. R. Donnelley in Crawfordsville, Indiana.

First printing, August 2006

Dedication

Jeffrey would like to dedicate this book to his wife Stephanie, and his three children Maeve, Aidan, and Rachel, for their love and support.

Jim would like to dedicate this book to his wife Beth, his parents Jim and Diane, Rupert Perera, and Larry Nordell for the immeasurable investments each of them has made in his life.

PREV
NEXT

About the Authors

Jeffrey Travis has extensive experience with software development, Web applications, Internet technologies, virtual instrumentation, and LabVIEW. He has provided LabVIEW and enterprise software consulting to both small and Fortune 100 companies looking for customized solutions in the areas of Web applications, LabVIEW programs, and remote instrumentation systems. He has been a guest speaker at symposiums and conferences on instrumentation and Internet technologies, and has published award-winning articles in technical journals. Jeffrey Travis is also the author of *Internet Applications in LabVIEW* (Prentice Hall, 2000) and the "LabVIEW Internet Applications" course.

Jeffrey is also an award-winning filmmaker and screenplay writer. His most recent work includes co-writing and directing the film *FLATLAND* based on Edwin A. Abbott's classical sci-fi novel.

Jeffrey currently lives in Austin, TX, with his wife and three children.

Jim Kring is an avid LabVIEW software engineer. He is also the CEO and founder of JKI, a LabVIEW and systems integration consulting firm that provides professional services and develops commercial software tools for professional LabVIEW developers. He believes that graphical software development will soon become a software industry standard and is avidly pushing the technology in that direction. Jim is a leader of the OpenG.org open source LabVIEW community, and is an active LabVIEW community member in all regards. He is changing the world, one VI at a time.

Jim has a BS in environmental engineering science from U.C. Berkeley, College of Engineering. He chose that major because it offered the widest variety of stimulating technical courses. He chose LabVIEW as his profession because it offers the widest variety of stimulating technical projects. He is a Certified LabVIEW Architect, has been a Certified LabVIEW Instructor, and has served as an instructor for LabVIEW and electronics courses at the collegiate level.

He was the lead architect of software development and systems integration for a commercial product that won the R&D 100 award, and he has received several awards for various technical publications.

Jim lives in San Francisco, CA, with his wife.

Preface

LabVIEW is a graphical programming language that has been widely adopted throughout industry, academia, and research labs as the standard for data acquisition and instrument control software. LabVIEW is a powerful and flexible instrumentation and analysis software system that is multiplatformyou can run LabVIEW on Windows, Mac OS X, and Linux. You can also run LabVIEW on PDAs (PalmOS, PocketPC, or Windows CE devices), on real-time platforms, and even embed LabVIEW programs into FPGA chips and 32-bit microprocessors. Creating your own LabVIEW program, or virtual instrument (VI), is simple. LabVIEW's intuitive user interface makes writing and using programs exciting and fun!

LabVIEW departs from the sequential nature of traditional programming languages and features an easy-to-use graphical programming environment, including all of the tools necessary for data acquisition (DAQ), data analysis, and presentation of results. With its graphical programming language, sometimes called "G," you program using a graphical block diagram that compiles into machine code. Ideal for a countless number of science and engineering applications, LabVIEW helps you solve many types of problems in only a fraction of the time and hassle it would take to write "conventional" code.

Beyond the Lab

LabVIEW has found its way into such a broad spectrum of virtual instrumentation applications that it is hard to know where to begin. As its name implies, it began in the laboratory and still remains very popular in many kinds of laboratoriesfrom major research and development laboratories around the world (such as Lawrence Livermore, Argonne, Batelle, Sandia, Jet Propulsion Laboratory, White Sands, and Oak Ridge in the United States, and CERN in Europe), to R&D laboratories in many industries, and to teaching laboratories in universities all over the world, especially in the disciplines of electrical and mechanical engineering and physics.

The spread of LabVIEW beyond the laboratory has gone in many directionsup (aboard the space shuttle), down (aboard U.S. Navy submarines), and around the world (from oil wells in the North Sea to factories in New Zealand). And with the latest Internet capabilities, LabVIEW applications are being deployed not only physically in many places, but virtually across networked applications. More and more people are creating web-based control or monitoring of their LabVIEW applications to allow remote access and instant information about what's happening in their lab. Virtual instrumentation systems are known for their low cost, both in hardware and development time, and their great flexibility.

The Expanding World of Virtual Instrumentation

Perhaps the best way to describe the expansion (or perhaps explosion) of LabVIEW applications is to generalize it. There are niches in many industries where measurements of some kind are requiredmost often of temperature, whether it be in an oven, a refrigerator, a greenhouse, a clean

room, or a vat of soup. Beyond temperature, users measure pressure, force, displacement, strain, pH, and so on, ad infinitum. Personal computers are used virtually everywhere. LabVIEW is the catalyst that links the PC with measuring things, not only because it makes it easy, but also because it brings along the ability to analyze what you have measured and display it and communicate it halfway around the world if you so choose.

After measuring and analyzing something, the next logical step often is to change (control) something based upon the results. For example, measure temperature and then turn on either a furnace or a chiller. Again, LabVIEW makes this easy to do; monitoring and control have become LabVIEW strengths. Sometimes it is direct monitoring and control, or it may be through communicating with a programmable logic controller (PLC) in what is commonly called supervisory control and data acquisition (SCADA).

The Results

A few of LabVIEW's many uses include the following:

- Simulating heart activity
- Controlling an ice cream-making process
- Detecting hydrogen gas leaks on the space shuttle
- Monitoring feeding patterns of baby ostriches
- Modeling power systems to analyze power quality
- Measuring physical effects of exercise in lab rats
- Controlling motion of servo and stepper motors
- Testing circuit boards in computers and other electronic devices
- Simulating motion in a virtual reality system
- Allowing remote navigation and feedback over the web of a helium-filled blimp
- Automatically generating cover sheets for your TPS reports

Objectives of This Book

LabVIEW for Everyone will help you get LabVIEW up and running quickly and easily, and will start you down the road to becoming an expert LabVIEW developer. The book offers additional examples and activities to demonstrate techniques, identifies other sources of information about LabVIEW, and features descriptions of cool LabVIEW applications. You are invited to open, inspect, use, and modify any of the programs on the accompanying CD-ROM. You can also get updates to the examples, activities, book errata, and other related resources and information at http://labviewforeveryone.com. The CD-ROM also includes the 30-day evaluation version of LabVIEW 8.0 for Windows, which allows you to do just about everything the commercial version does during

the evaluation period. You can also always get the latest evaluation version of LabVIEW at http://ni.com/labview.

This book expects you to have basic knowledge of your computer's operating system. If you don't have much computer experience, you may want to spend a little time with your operating system manual and familiarize yourself with your computer. For example, you should know how to access menus, open and save files, make backup disks, and use a mouse. It also helps if you have some basic programming experience with other languages (C, Java, FORTRAN, etc.), but it is not necessary to know another programming language to use LabVIEW.

After reading this book and working through the exercises, you should be able to do the following, and much more, with the greatest of ease:

- Write LabVIEW programs, called virtual instruments, or VIs.
- Employ various debugging techniques.
- Manipulate both built-in LabVIEW functions and VIs.
- Create and save your own VIs so that you can use them as subVIs, or subroutines.
- Design custom graphical user interfaces (GUIs).
- Save your data in a file and display it on a graph or chart.
- Build applications that use General Purpose Interface Bus (GPIB) or serial instruments.
- Create applications that use plug-in DAQ boards.
- Use built-in analysis functions to process your data.
- Optimize the speed and performance of your LabVIEW programs.
- Employ advanced techniques such as state machines and event structures.
- Control your VIs and publish your data over the Internet or on the Web, using LabVIEW's features like its built-in Web server and remote panels.
- Use LabVIEW to create your instrumentation applications.

LabVIEW for Everyone helps you get started quickly with LabVIEW to develop your instrumentation and analysis applications. The book is divided into two main sections: *Fundamentals* and *Advanced Topics*.

The *Fundamentals* section contains nine chapters and teaches you the fundamentals of programming in LabVIEW. The *Advanced Topics* section contains eight chapters that further develop your skills and introduce helpful techniques and optimizing strategies. We suggest that you work through the beginning section to master the basics; then, if you're short on time, skip around to what you really want to learn in the advanced section.

In both sections, chapters have a special structure to facilitate learning, as follows:

- *Overview*, *goals*, and *key terms* describe the main ideas covered in that chapter.
- The main sections are a discussion of the featured topics.
- *Activities* reinforce the information presented in the discussion.
- *Wrap It Up!* summarizes important concepts and skills taught in the chapter.
- Additional activities in many chapters give you more practice with the new material.

Fundamentals

Chapter 1, "What in the World is LabVIEW?," describes LabVIEW and introduces you to some of LabVIEW's features and uses.

In Chapter 2, "Virtual Instrumentation: Hooking Your Computer Up to the Real World," you will get an overview of virtual instrumentation: how data acquisition, GPIB, serial port communication, and data analysis are performed with LabVIEW.

In Chapter 3, "The LabVIEW Environment," you will get acquainted with the LabVIEW environment, including the LabVIEW Project Explorer, the essential parts of a virtual instrument (or VI), the Help window, menus, tools, palettes, and subVIs.

In Chapters 4 and 5, "LabVIEW Foundations" and "Yet More Foundations," you will become familiar with the basics of programming in LabVIEWusing controls and indicators (such as numerics, Booleans, and strings); wiring, creating, editing, debugging, and saving VIs; creating subVIs; and documenting your work. You will also begin to understand why LabVIEW is considered a dataflow programming language.

Chapter 6, "Controlling Program Execution with Structures," describes the basic programming structures in LabVIEW: While Loops, For Loops, shift registers, Case Structures, Sequence Structures, and Formula Nodes. It also teaches you how to introduce timing into your programs. You will be introduced to easy-to-use frameworks that combine the While Loop and Case Structure to build scalable applications.

In Chapter 7, "LabVIEW's Composite Data: Arrays and Clusters," you will learn how to use two important data structuresarrays and clustersin your programs. You will also explore LabVIEW's built-in functions for manipulating arrays and clusters. It also teaches you about the error cluster and how to perform proper error handling in LabVIEW.

Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs," details the variety of charts and graphs available in LabVIEW and teaches you how to use them for animated and informative data presentation. It also introduces the waveform, time stamp, and dynamic datatypes.

Chapter 9, "Exploring Strings and File I/O," discusses string data types, string functions, and tables. You will learn how to parse strings like a pro, using the regular expression syntax. It also talks a little about how to save data in and read data from a file, using LabVIEW's simple File I/O VIs.

Advanced Topics

Chapter 10, "Signal Measurement and Generation: Data Acquisition," teaches you some of the theory behind signals, data acquisition, and analog and digital I/O. You will learn some hardware considerations, and you will find a valuable guide to many common acronyms used in instrumentation. Chapter 10 also discusses software setup for data acquisition using the Measurement & Automation Explorer (MAX) utility and configuring NI-DAQmx devices.

Chapter 11, "Data Acquisition in LabVIEW," then takes the basics of data acquisition you learned in Chapter 10 and teaches you how to apply them in LabVIEW using the DAQmx VIs. You'll learn about the easy-to-use and powerful DAQmx tasks in LabVIEW, as well as work through some activities to read and write analog data, digital data. The chapter ends with some more advanced forms of data acquisition applications, such as streaming data to a file or doing a triggered acquisition.

Chapter 12, "Instrument Control in LabVIEW," discusses how to use external instruments with LabVIEW. You'll learn about GPIB, serial, Ethernet, and other kinds of instrument interfaces, and how to use the powerful VISA framework with LabVIEW to easily communicate with them.

Chapter 13, "Advanced LabVIEW Structures and Functions," covers some invaluable features like local and global variables, property nodes, invoke nodes, and the powerful Event Structure for event-driven programming. You will learn about the State Machine and Queued Message Handler application frameworks, as well as the Messaging and Synchronization functions: Queues, Notifiers, Semaphores, Rendezvous, and Occurrences. It also introduces you to some more of LabVIEW's cool GUI widgets like trees, subpanels, graphics and sound, and so on.

Chapter 14, "Advanced LabVIEW Data Concepts," discusses some more file I/O, showing you how to work with binary files and configuration files, as well as advanced text file functions. You'll see how LabVIEW can both read and generate external code modules such as DLLs and Shared Libraries.

Chapter 15, "Advanced LabVIEW Features," shows you how to configure VI behavior and appearance using VI Setup options. You'll learn about the powerful VI Server and how you can dynamically control LabVIEW, VIs, and controls by reference. It also introduces LabVIEW's very useful tools such as the Find function and VI Hierarchy window.

Chapter 16, "Connectivity in LabVIEW," covers connectivity features in LabVIEW such as printing to the web, controlling VIs remotely over the web, sharing data with Shared Variables, networking, and report generation.

In Chapter 17, "The Art of LabVIEW Programming," you will learn about good LabVIEW style and some new tips, such as how to add a customized look to your applications by importing pictures and using the Control Editor. Chapter 17 describes some good programming techniques that you can use to make your programs run faster, use less memory, port more easily to other platforms, and behave more efficiently overall.

In Appendix A, "CD Contents," you can find an outline and description of the files contained on the accompanying CD-ROM, as well as high-level instructions for installing the LabVIEW evaluation version and finding the examples and activities discussed in this book.

In Appendix B, "Add-on Toolkits for LabVIEW," you will learn about add-on toolkits available from National Instruments and third-party companies to increase LabVIEW's functionality.

Appendix C, "Open Source Tools for LabVIEW: OpenG," introduces you to free (as in speech) software and the OpenG.org community that collaboratively develops add-on toolkits for LabVIEW.

In Appendix D, "LabVIEW Object-Oriented Programming," you will be introduced to object-oriented

programming techniques in LabVIEW, including a history of LabVIEW object-oriented programming and some perspectives on things to come.

Appendix E, "Resources for LabVIEW," contains links to various LabVIEW resources such as user groups, discussion forums, and various other online LabVIEW resources.

In Appendix F, "LabVIEW Certification Exams," you will learn about the LabVIEW developer certification exams, how to prepare for them, what to expect, and the benefits of certification.

You will find a glossary and index at the end of the book.

The following table describes the conventions used in this book.

bold	Bold text denotes VI names, function names, menus, menu items, and palettes. In addition, bold text denotes VI input and output parameters. For example, "Choose TCP Read from the TCP Functions palette."
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key term or concept. For example, "A *control reference* is an object that points to a LabVIEW control or indicator and can manage its properties."
`Courier`	`Courier` type denotes text or characters that you enter using the keyboard. It also denotes files and paths. Sections of code, programming examples, syntax examples, and messages and responses that the computer automatically prints to the screen also appear in this font. For example, "In the text book, enter `c:\data\datafile.txt` as the filename."
	Note. This icon marks information to which you should pay special attention.
	Watch Out! This icon flags a common pitfall or special information that you should be aware of in order to keep out of trouble.
	Hint. This icon calls your attention to useful tips and hints on how to do something efficiently.
	When you see this icon, it means the topic being discussed is a Certified LabVIEW Developer (CLD) exam topic. If you want to prepare to become a CLD, pay attention! See Appendix F for detailed information about the certification program, what you can gain by becoming certified, how to study, and what to expect while taking the exams.

	In Chapter 6, you will be introduced to LabVIEW's Express technologies, which provide configurable automatic programming for common tasks. This icon indicates the section refers to an Express Technology topic.

A Note about Paths

Different platforms have different conventions for specifying path names. For example, Windows paths take the form `X:\LABVIEW\MINE.LLB\BINGO.VI.` The same path on a MacOS classic system would be denoted `Hard Drive Name:LabVIEW:Mine.jib:Bingo.vi.` On Linux or OS X machines, it would be `/usr/labview/mine.llb/bingo.vi.` Rather than specifying a full path, this book will list the default path from the LabVIEW directory or folder when telling you where to find an example VI. To simplify notation, we will use the Windows standard to describe paths; if you use Mac OS or UNIX machines, please substitute colons or forward slashes where necessary.

What's New in the Third Edition

LabVIEW For Everyone was the first book published aimed at the beginner LabVIEW user in 1997 for LabVIEW 4.0; since then, Prentice Hall and other publishers have produced over a dozen LabVIEW books on specific topics. The second edition of *LabVIEW For Everyone* was updated for LabVIEW 6.1 and introduced some new topics.

This third edition has been completely revised and updated for LabVIEW 8.0. Two new chapters were added, and the number of pages has almost doubled! The changes from the second edition include the following:

- Inclusion of the new Express VIs and Express technology in LabVIEW
- Inclusion of the Certified LabVIEW Developer callouts to identify key sections that provide information that is tested on the CLD certification exam
- Autotool
- Static VI reference
- Call by Reference advanced options
- Event structure and event-driven programming
- Dynamic data
- Variants
- Type definitions
- Configuration (INI) file VIs

- Calling DLLs from LabVIEW
- Shared variables
- Custom probes
- Search and replace
- XML
- Pipes
- Error handling design patterns, tips, and tricks
- While Loop + case structure design patterns
- State machine and queued message handler design patterns
- Messaging and Synchronization using Queues, Notifiers, Semaphores, Rendezvous, and Occurrences
- Tree controls, tab controls, drag and drop, subpanels, scrollbars, and splitter bars
- Regular expressions
- Diagram disable and conditional disable structures
- Using NI-DAQmx and the LabVIEW DAQmx VIs
- LabVIEW project explorer
- Alignment grid
- Timed structures (timed loop and timed sequence structure)
- Advanced report generation
- Polymorphic VIs

LabVIEW Installation Instructions

If you own a licensed version of LabVIEW and need instructions on how to install it, please see the release notes that came with your software.

Otherwise, you can install a 30-day evaluation version of LabVIEW. Although we've included an evaluation version installer on this book's CD, we encourage you to check http://ni.com/labview for the latest evaluation version of LabVIEW, available as a download. The evaluation version of LabVIEW included on this CD, version 8.0 for Windows, will allow you to work through all the examples in this book. Remember, the evaluation version stops working after 30 days. After that, you can activate your installed LabVIEW to a licensed version by purchasing a license from National Instruments at http://ni.com.

In addition, you will need to access the EVERYONE directory from the CD-ROM in the back of this book. It contains the activities in this book and their solutions. You may want to copy the EVERYONE directory onto your PC so you can save your activity work there as well. (You can also get updates to the examples, activities, book errata, and other related resources and information at http://labviewforeveryone.com.)

Purchasing LabVIEW

If you would like information on how to purchase LabVIEW, contact National Instruments:

```
National Instruments
11500 N Mopac Expwy
Austin, TX 78759-3504
Telephone: 888-280-7645
Fax: 512-683-8411
Email: info@ni.com
Web: http://www.ni.com
```

PREV NEXT

Acknowledgments

The authors would like to acknowledge the contributions of the following individuals for their assistance with making this third edition of *LabVIEW for Everyone* possible:

To Bernard Goodwin, our editor at Pearson, who encouraged us to write this updated third edition and who held us to our deadlines.

To the fine folks at National Instruments who assisted with the book by providing resources, technical reviews, artwork, and their time. In particular, we'd like to especially thank Craig Anderson, Jim Cahow, Zaki Chasmawala, David Corney, Crystal Drumheller, David Gardner, Stephen Mercer, Darren Natinger, Jeff Peters, and Brian Tyler.

To the LabVIEW gurus who reviewed our manuscript, and caught all our mistakes (we hope!) before it went to print: Mike Ashe, John Compton-Smith, Ed Dickens, Jean-Pierre Drolet, Crystal Drumheller, Kostya Shifershteyn, Paul Sullivan, and Ashish Uttarwar.

To Michael Aivaliotis for maintaining the LAVA (LabVIEW Advanced Virtual Architects) discussion forums on the Web (forums.lavag.org) and the community of LAVA members who make this forum so great. We regularly used this incredibly useful resource throughout the writing of this third edition for discussing technical topics and polling the community of LabVIEW users on key topics.

Jim would like to especially acknowledge, first and foremost, his wife Beth, who not only put up with, but also provided tremendous support in, his writing of this third edition. And also to his family, friends, and co-workers for their support and understanding during the many long months he spent in seclusion while working on the manuscript and for welcoming him back upon its completion.

1. What in the World Is LabVIEW?

Overview

Welcome to the world of LabVIEW! This chapter gives you a basic explanation of LabVIEW, its capabilities, and how it can make your life easier.

Goals

- Develop an idea of what LabVIEW really is.
- Learn what "graphical programming language" and "dataflow programming" mean.
- Peruse the introductory examples that come installed with LabVIEW using the NI Example Finder.
- Get a feel for the LabVIEW environment.

Key Terms

- LabVIEW
- NI Example Finder
- G
- Virtual instrument (VI)
- Dataflow
- Graphical language
- Front panel
- Block diagram
- Icon
- Connector
- Toolbar
- Palette
- Hierarchy

What Exactly Is LabVIEW, and What Can It Do for Me?

LabVIEW, short for *Laboratory Virtual Instrument Engineering Workbench*, is a programming environment in which you create programs using a graphical notation (connecting functional nodes via wires through which data flows); in this regard, it differs from traditional programming languages like C, C++, or Java, in which you program with text. However, LabVIEW is much more than a programming language. It is an interactive program development and execution system designed for people, like scientists and engineers, who need to program as part of their jobs. The LabVIEW development environment works on computers running Windows, Mac OS X, or Linux. LabVIEW can create programs that run on those platforms, as well as Microsoft Pocket PC, Microsoft Windows CE, Palm OS, and a variety of embedded platforms, including Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), and microprocessors.

Using the very powerful graphical programming language that many LabVIEW users affectionately call "G" (for graphical), LabVIEW can increase your productivity by orders of magnitude. Programs that take weeks or months to write using conventional programming languages can be completed in hours using LabVIEW because it is specifically designed to take measurements, analyze data, and present results to the user. And because LabVIEW has such a versatile graphical user interface and is so easy to program with, it is also ideal for simulations, presentation of ideas, general programming, or even teaching basic programming concepts.

LabVIEW offers more flexibility than standard laboratory instruments because it is software-based. You, not the instrument manufacturer, define instrument functionality. Your computer, plug-in hardware, and LabVIEW comprise a completely configurable virtual instrument to accomplish your tasks. Using LabVIEW, you can create exactly the type of virtual instrument you need, when you need it, at a fraction of the cost of traditional instruments. When your needs change, you can modify your virtual instrument in moments.

LabVIEW tries to make your life as hassle-free as possible. It has extensive libraries of functions and subroutines to help you with most programming tasks, without the fuss of pointers, memory allocation, and other arcane programming problems found in conventional programming languages. LabVIEW also contains application-specific libraries of code for data acquisition (DAQ), General Purpose Interface Bus (GPIB), and serial instrument control, data analysis, data presentation, data storage, and communication over the Internet. The Analysis Library contains a multitude of useful functions, including signal generation, signal processing, filters, windows, statistics, regression, linear algebra, and array arithmetic.

Figure 1.1. The Space Industries Sheet Float Zone Furnace is used for high-temperature superconductor materials processing research in a microgravity environment aboard the NASA KC-135 parabolic aircraft. LabVIEW controls the industrialized Mac OS-based system.

Because of LabVIEW's graphical nature, it is inherently a data presentation package. Output appears in any form you desire. Charts, graphs, and user-defined graphics comprise just a fraction of available output options. This book will show you how to present data in all of these forms.

LabVIEW's programs are portable across platforms, so you can write a program on a Macintosh and then load and run it on a Windows machine without changing a thing in most applications. You will find LabVIEW applications improving operations in any number of industries, from every kind of engineering and process control to biology, farming, psychology, chemistry, physics, teaching, and many others.

Dataflow and the Graphical Programming Language

The LabVIEW program development environment is different from standard C or Java development systems in one important respect: While other programming systems use text-based languages to create lines of code, LabVIEW uses a graphical programming language, often called "G," to create programs in a pictorial form called a *block diagram*.

Graphical programming eliminates a lot of the syntactical details associated with text-based languages, such as where to put your semicolons and curly braces. (If you don't know how text-based languages use these, don't worry. With LabVIEW, you don't need to know!)

Graphical programming allows you to concentrate on the flow of data within your application, because its simple syntax doesn't obscure what the program is doing. Figures 1.2 and 1.3 show a simple LabVIEW user interface and the code behind it.

Figure 1.2. User interface

[View full size image]

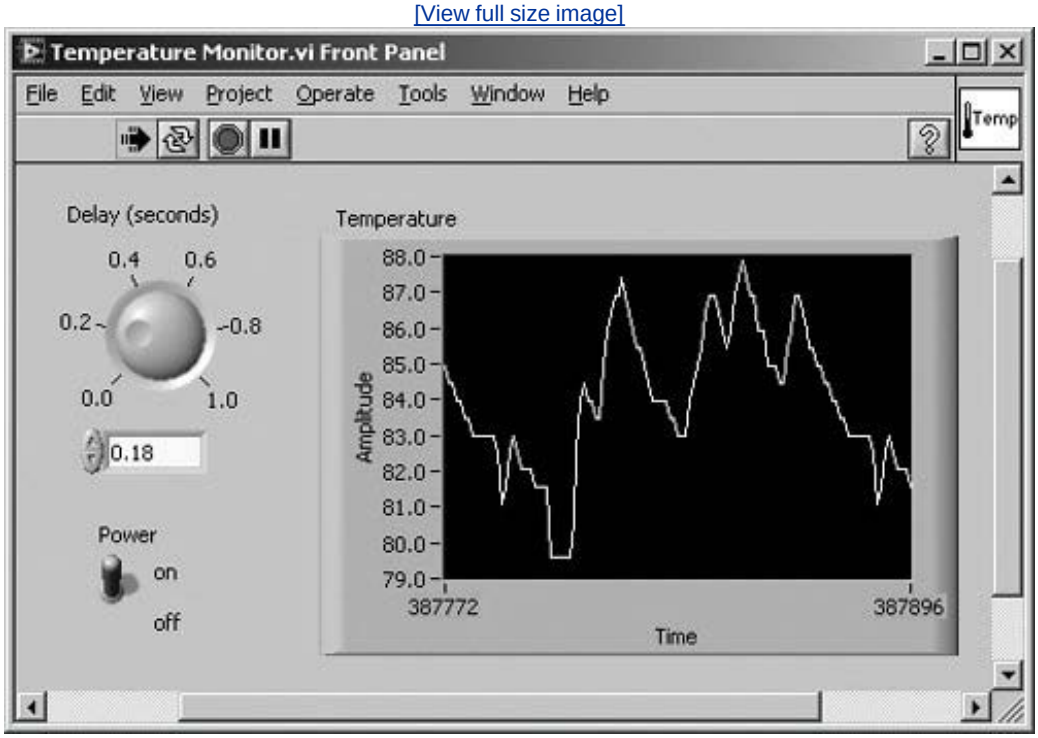
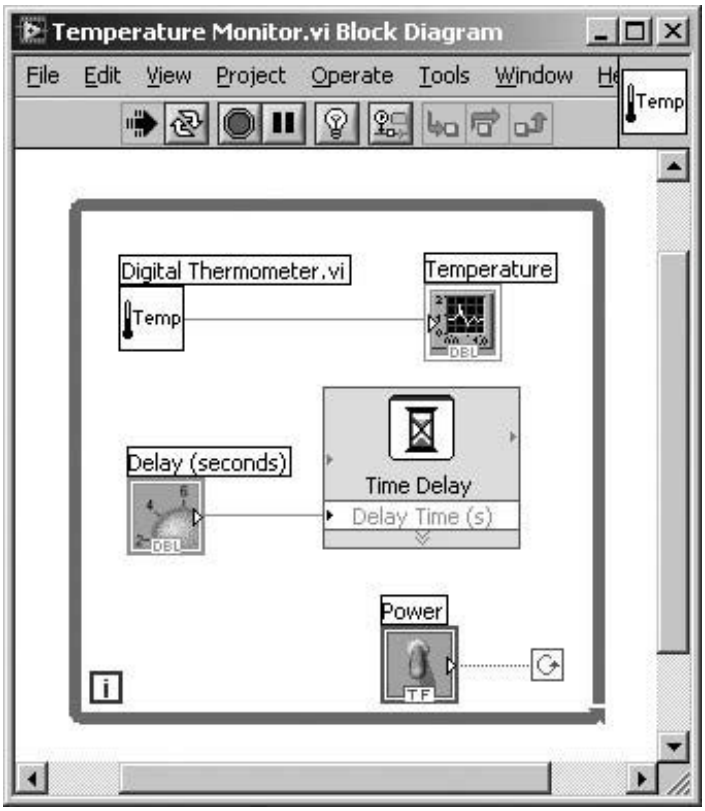


Figure 1.3. Graphical code



LabVIEW uses terminology, icons, and ideas familiar to scientists and engineers. It relies on graphical symbols rather than textual language to define a program's actions. Its execution is based on the principle of dataflow, in which functions execute only after receiving the necessary data. Because of these features, you can learn LabVIEW even if you have little or no programming experience. However, you will find that a knowledge of programming fundamentals is very helpful.

How Does LabVIEW Work?

A LabVIEW program consists of one or more virtual instruments (VIs). Virtual instruments are called such because their appearance and operation often imitate actual physical instruments. However, behind the scenes, they are analogous to main programs, functions, and subroutines from popular programming languages like C or Basic. Hereafter, we will refer to a LabVIEW program as a "VI" (pronounced "vee eye," NOT the Roman numeral six, as we've heard some people say). Also, be aware that a LabVIEW program is always called a VI, whether its appearance or function relates to an actual instrument or not.

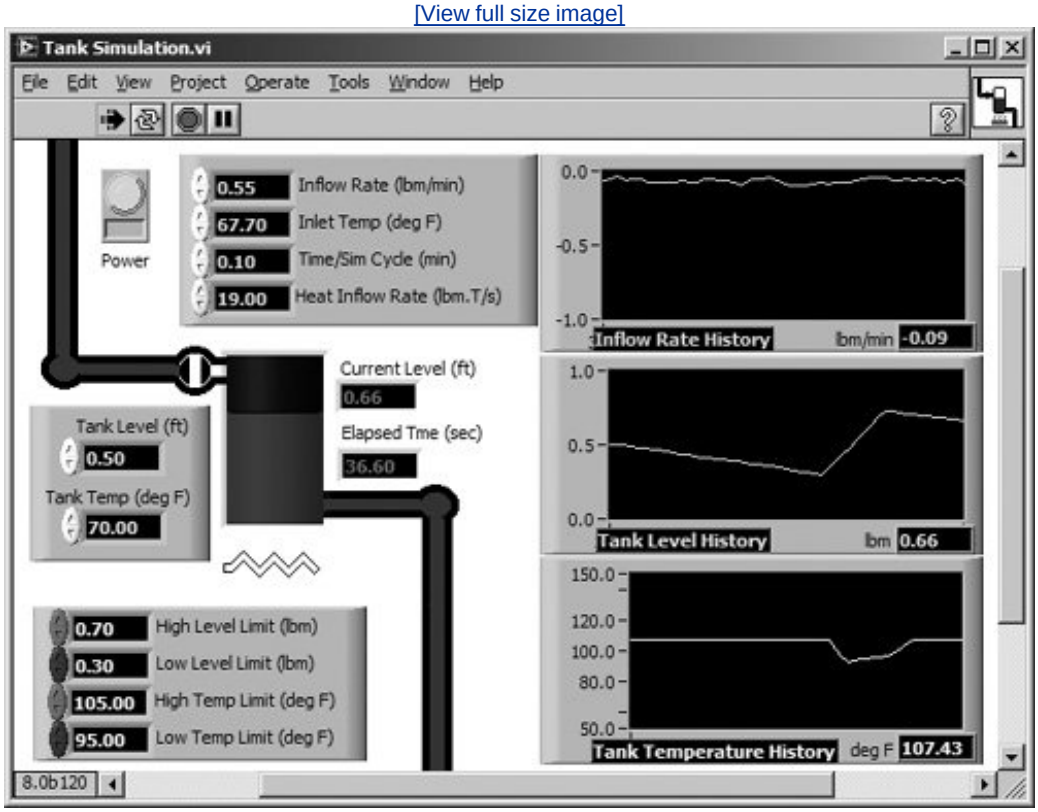
A VI has three main parts: a front panel, a block diagram, and an icon.

- The *front panel* is the interactive user interface of a VI, so named because it simulates the front panel of a physical instrument (see Figure 1.4). The front panel can contain knobs, push

buttons, graphs, and many other controls (which are user inputs) and indicators (which are program outputs). You can input data using a mouse and keyboard, and then view the results produced by your program on the screen.

Figure 1.4. A VI front panel

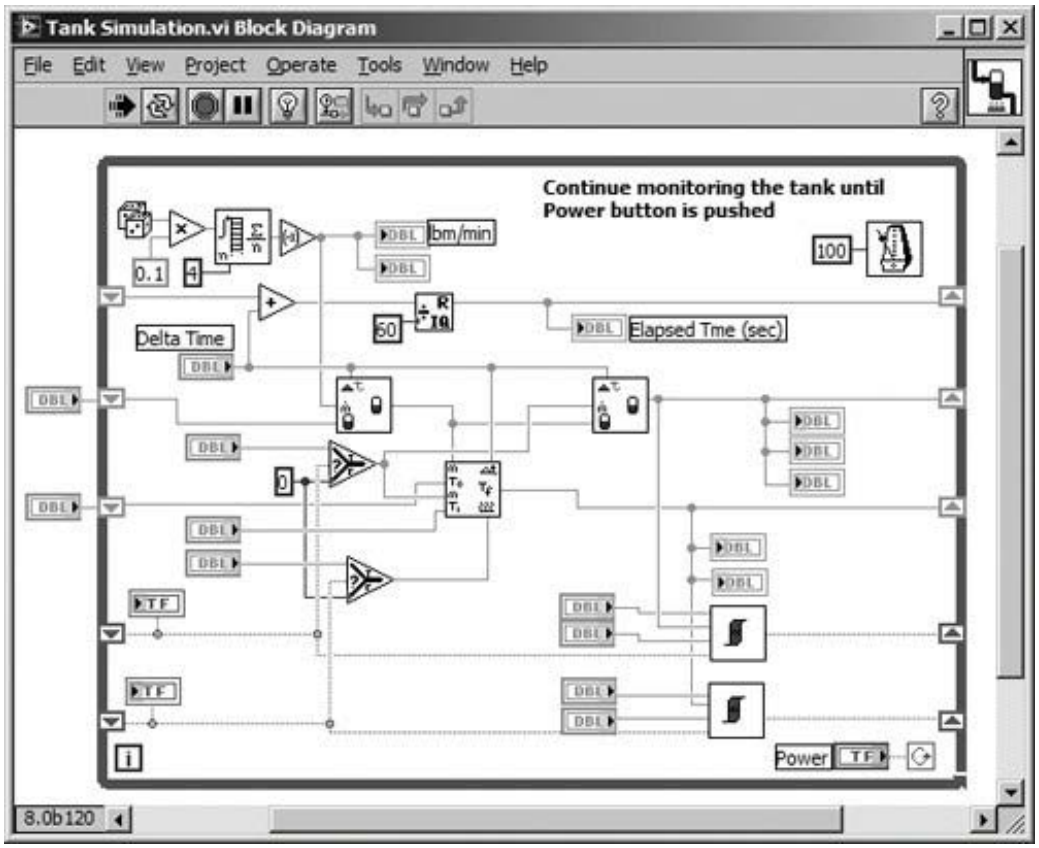
[View full size image]



- The *block diagram* is the VI's source code, constructed in LabVIEW's graphical programming language, G (see Figure 1.5). The block diagram is the actual executable program. The components of a block diagram are lower-level VIs, built-in functions, constants, and program execution control structures. You draw wires to connect the appropriate objects together to define the flow of data between them. Front panel objects have corresponding terminals on the block diagram so data can pass from the user to the program and back to the user.

Figure 1.5. A VI block diagram

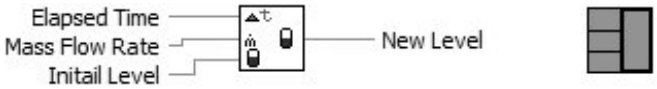
[View full size image]



- 

  In order to use a VI as a subroutine in the block diagram of another VI, it must have an *icon* with a *connector* (see Figure 1.6). A VI that is used within another VI is called a *subVI* and is analogous to a subroutine. The icon is a VI's pictorial representation and is used as an object in the block diagram of another VI. A VI's connector is the mechanism used to wire data into the VI from other block diagrams when the VI is used as a subVI. Much like parameters of a subroutine, the connector defines the inputs and outputs of the VI.

Figure 1.6. VI icon (left) and connector (right)



Virtual instruments are hierarchical and modular. You can use them as top-level programs or subprograms. With this architecture, LabVIEW promotes the concept of modular programming. First, you divide an application into a series of simple subtasks. Next, you build a VI to accomplish each subtask and then combine those VIs on a top-level block diagram to complete the larger task.

Modular programming is a plus because you can execute each subVI by itself, which facilitates debugging. Furthermore, many low-level subVIs often perform tasks common to several applications and can be used independently by each individual application.

Just so you can keep things straight, we've listed a few common LabVIEW terms with their conventional programming equivalents in Table 1.1.

Table 1.1. LabVIEW Terms and Their Conventional Equivalents

LabVIEW	*Conventional Language*
VI	program
function	function or method
subVI	subroutine, subprogram, object
front panel	user interface
block diagram	program code
G	C, C++, Java, Pascal, BASIC, etc.

If you've worked with object-oriented languages before such as C++ or Java, you should know that LabVIEW and G in it simplest form is not truly an object-oriented language. However, object-oriented programming can provide many benefits, which is why there are several toolkits that let you write object-oriented code in G, known as GOOP (G Object-

PREV NEXT

Demonstration Examples

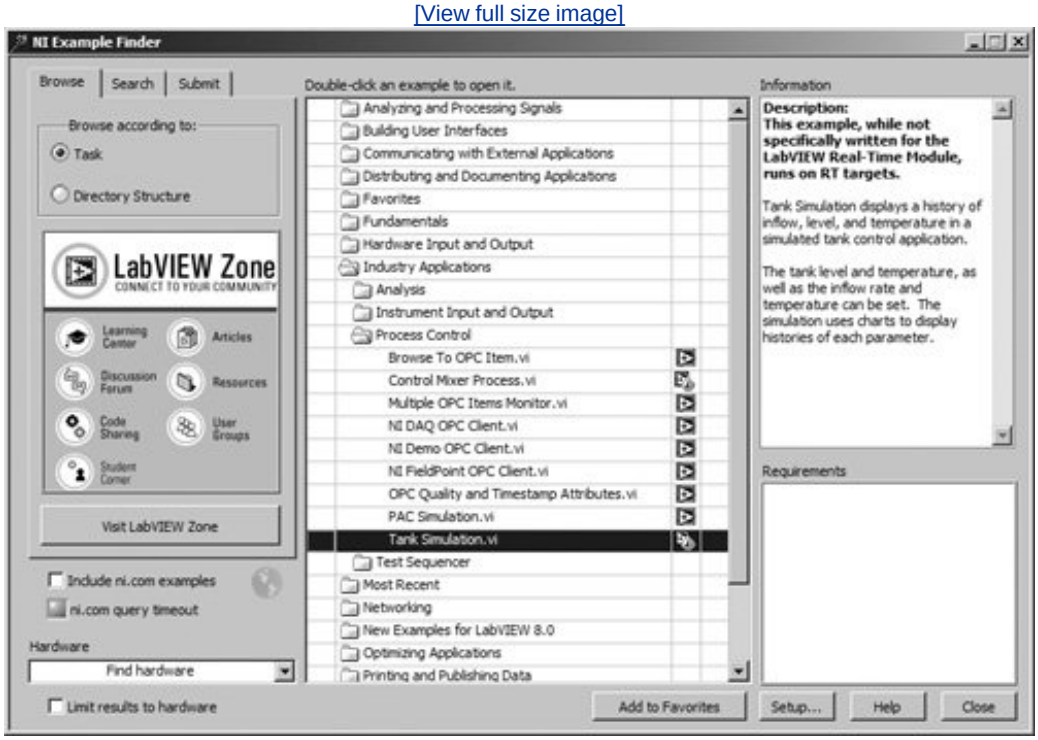
Okay, you have had enough reading for now. To get an idea of how LabVIEW works, you can open and run a few existing LabVIEW programs.

NI Example Finder

LabVIEW ships with many working examples that will help you learn common programming techniques and see applications that accomplish common hardware input/output and data processing tasks. The NI Example Finder is a very useful tool that assists in the search for these examples. You can open the NI Example Finder, shown in Figure 1.7, by launching LabVIEW and then going to the Help menu and selecting Find Examples . . . .

Figure 1.7. The NI Example Finder

[View full size image]



If you are just getting started with LabVIEW, you will want to set the Browse according to: option to Task and start exploring inside the Industry Applications folder. This folder contains

demonstration and simulation applications, which are an excellent place to begin learning about LabVIEW. If you are looking for an example on a topic that you don't see listed in the folder tree view, you can switch to the Search tab of the Example Finder and perform a keyword search.

The previous steps are the process for quickly loading example VIs that come with LabVIEW. You can also access all the LabVIEW example VIs directly in the examples directory inside your LabVIEW installation directory. For example, on Windows, LabVIEW is usually installed at `C:\Program Files\National Instruments\LabVIEW`. *So the examples directory is at* `C:\Program Files\National Instruments\LabVIEW\examples`. *The Temperature System Demo example in particular would be located at* `C:\Program Files\National Instruments\LabVIEW\examples\apps\tempsys.llb\Temperature System Demo.vi`.

Generally, however, it's easier to find examples by using the NI Example Finder feature as just described.

Examples on the CD

Whether you are using the Professional, Full, or Evaluation version of LabVIEW, just launch it. Make sure you can access the `EVERYONE` directory from the CD or your hard drive, as described in the Preface; it contains the activities for this book. After launching LabVIEW, a dialog box will appear. To open an example, select Open VI and choose the one you want.

Throughout this book, use the left mouse button (if you have more than one) unless we specifically tell you to use the right one (often we will tell you to "right-click" on something). On Mac OS X computers that have a one-button mouse, <control>-click when right-mouse functionality is necessary.

Activity 1-1: Temperature System Demo

Open and run the VI called Temperature System Demo.vi by following these steps:

1. Launch the NI Example Finder, as described in the NI Example Finder section of this chapter.

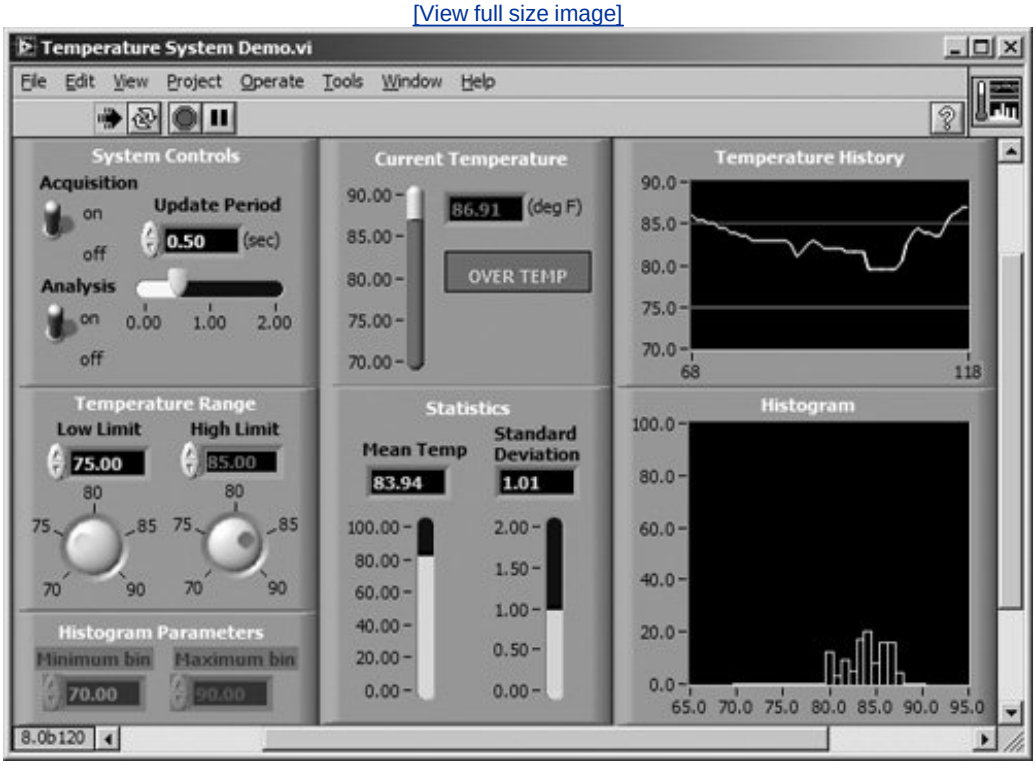
2. With the Browse tab selected and the Browse according to: option set to Task, navigate the folder tree to "Industry Applications," then "Analysis." Double-click "Temperature System Demo.vi" to open it. This VI may also be found using the File>>Open menu option and browsing to the following location, beneath the LabVIEW installation folder:

   `examples/apps/tempsys.llb/Temperature System Demo.vi`

3. You will see the VI shown in Figure 1.8.

Figure 1.8. Temperature System Demo.vi front panel

[View full size image]



4. Run the VI by clicking on the Run button, located on the VI's Toolbar (the *Toolbar* is the row of icons beneath the VI's menubar). The Run button will change appearance to indicate that the VI is running. Other buttons on the Toolbar may also change appearance (or disappear) because certain buttons are only applicable while the VI is running (such as the Abort button), and others are only applicable while the VI is not running (such as those used for editing).

Run Button Inactive

Run Button Active


Abort Button

Notice also that the Abort button becomes active in the Toolbar. You can press it to abort program execution.

Temperature System Demo.vi simulates a temperature monitoring application. The VI makes temperature measurements and displays them in the thermometer indicator and on the chart. Although the readings are simulated in this example, you can easily modify the program to measure real values. 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; you can change these limits using the Temperature Range knobs. 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 Statistics section shows you a running calculation of the mean and standard deviation, and the Histogram plots the frequency with which each temperature value occurs.

Tweaking Values

5. Use the cursor, which takes on the personality of the Operating tool while the VI is running, to change the values of the high and low limits. Highlight the old high or low value, either by clicking twice on the value you want to change, or by clicking and dragging across the value with the Operating tool. Then type in the new value and click on the enter button, located next to the run button on the Toolbar. Also, try changing the high and low limits using the round knob controls. Note how changing the values with the knobs takes effect instantly.


Operating Tool

6. Change the Update Period slide control by placing the Operating tool on the slider, and then clicking and dragging it to a new location.

Enter Button

You can also operate slide controls using the Operating tool by clicking on a point on the slide to

snap the slider to that location, by clicking on a scroll button to move the slider slowly toward the arrow, or by clicking in the slide's digital display and entering a number.

Even though the display changes, LabVIEW does not accept the new values in digital displays until you press the enter button, or click the mouse in an open area of the window. This is different from the behavior of the knob, which updates the value immediately.

7. Try adjusting the other controls in a similar manner.

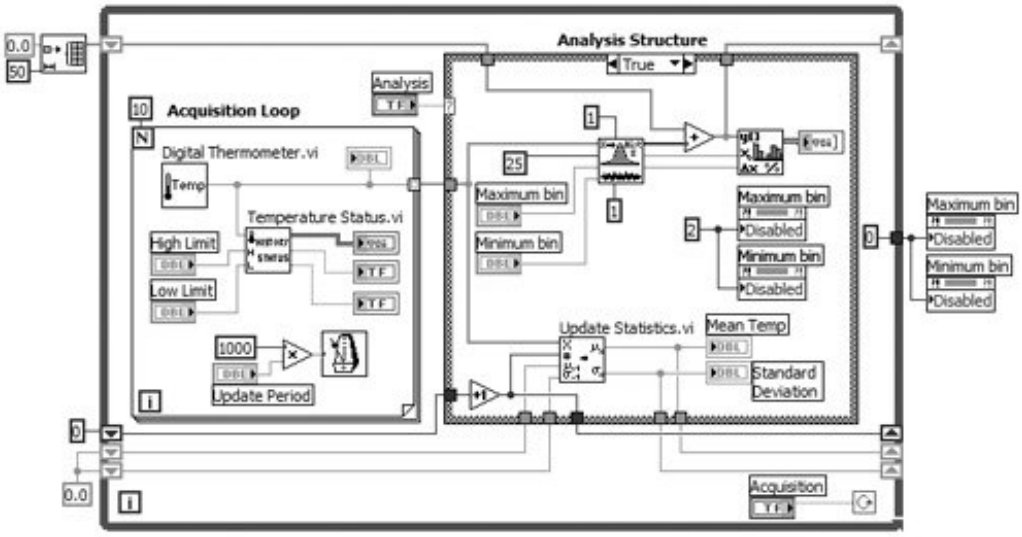
8. Stop the VI by clicking on the Acquisition switch.

Examine the Block Diagram

The block diagram shown in Figure 1.9 represents a complete LabVIEW application. You don't need to understand all of these block diagram elements right nowwe'll deal with them later. Just get a feel for the nature of a block diagram. If you already do understand this diagram, you'll probably fly through the first part of this book!

Figure 1.9. Temperature System Demo.vi block diagram

[View full size image]



9. Open the block diagram of Temperature System Demo.vi by choosing Show Diagram from the Windows menuor you can use the <ctrl>-E shortcut on Windows, <command>-E on Mac OS X, or <meta>-E on Linux.

10. Examine the different objects in the diagram window. Don't panic at the detail shown here! These structures are explained step by step later in this book.

11. Open the contextual Help window by choosing Show Context Help from the Help menuor you can use the <ctrl>-H shortcut on Windows, <command>-H on Mac OS X, or <meta>-H on Linux. Position the cursor over different objects in the block diagram and watch the Help window change to show descriptions of the objects. If the object is a function or subVI, the Help window will describe the inputs and outputs as well.


Highlight Execution Button

12. Turn on execution highlighting by clicking on the Highlight Execution button, so that the light bulb changes to the *active* (lighted) state. With execution highlighting turned on, you can watch the data flow through the wires. You will see small data bubbles that travel along the wires, representing the data flowing through the wires. We will learn more about this and other useful debugging tools in Chapter 5.


Highlight Execution Button (Active)

Hierarchy

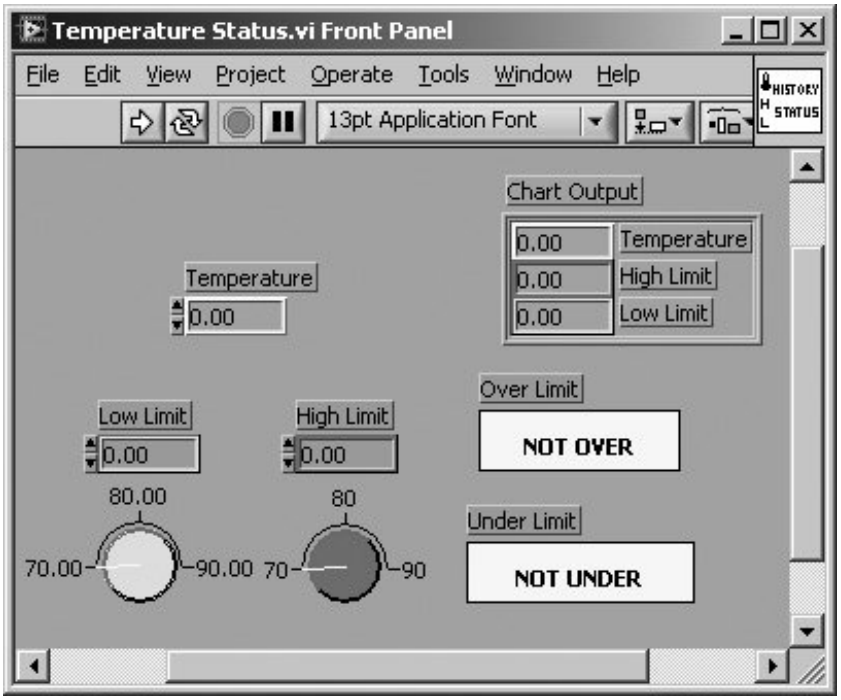
LabVIEW's power lies in the hierarchical nature of its VIs. After you create a VI, you can use it as a subVI in the block diagram of a higher-level VI, and you can have as many layers of hierarchy as you need. To demonstrate this versatile ability, look at a subVI of Temperature System Demo.vi.

13. Open the Temperature Status subVI by double-clicking on its icon.

Temperature Status subVI

The front panel shown in Figure 1.10 springs to life.

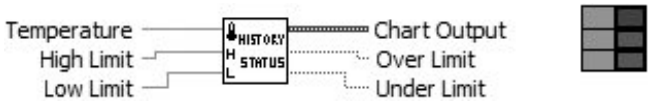
Figure 1.10. The front panel of the Temperature Status subVI



Icon and Connector

The icon and connector provide the graphical representation and parameter definitions needed if you want to use a VI as a sub-routine or function in the block diagrams of other VIs. They reside in the upper-right corner of the VI's front panel window. The icon graphically represents the VI in the block diagram of other VIs, while the connector terminals are where you must wire the inputs and outputs. These terminals are analogous to parameters of a subroutine or function. You need one terminal for each front panel control and indicator through which you want to pass data to the VI. The icon sits on top of the connector pattern until you choose to view the connector.

Figure 1.11. Temperature Status.vi Icon and Connector Pane



By using subVIs, you can make your block diagrams modular and more manageable. This modularity makes VIs easy to maintain, understand, and debug. In addition, you can often create one sub-VI to accomplish a function required by many different VIs.

Now run the top-level VI with both its window and the Temperature Status subVI window visible. Notice how the subVI values change as the main program calls it over and over.

14. Select Close from the File menu of the Temperature Status subVI. Do not save any changes.

15. Select Close from the File menu of Temperature System Demo.vi, and do not save any changes.

Selecting Close from the File menu of a VI diagram closes the block diagram window only. Selecting Close on a front panel window closes both the panel and the diagram.

Activity 1-2: Frequency Response Example

This example measures the frequency response of an unknown "black box." A function generator supplies a sinusoidal input to the black box (hint: it contains a bandpass filter, which lets only certain signal components through it). A digital multi-meter measures the output voltage of the black box. Although this VI uses subVIs to simulate a function generator and a digital multimeter, real instruments could easily be hooked up to a real black box to provide real-world data. You would then use sub-VIs to control data acquisition, GPIB transfers, or serial port communication to bring in or send out real data instead of simulating it.

You will open, run, and observe the VI in this activity.

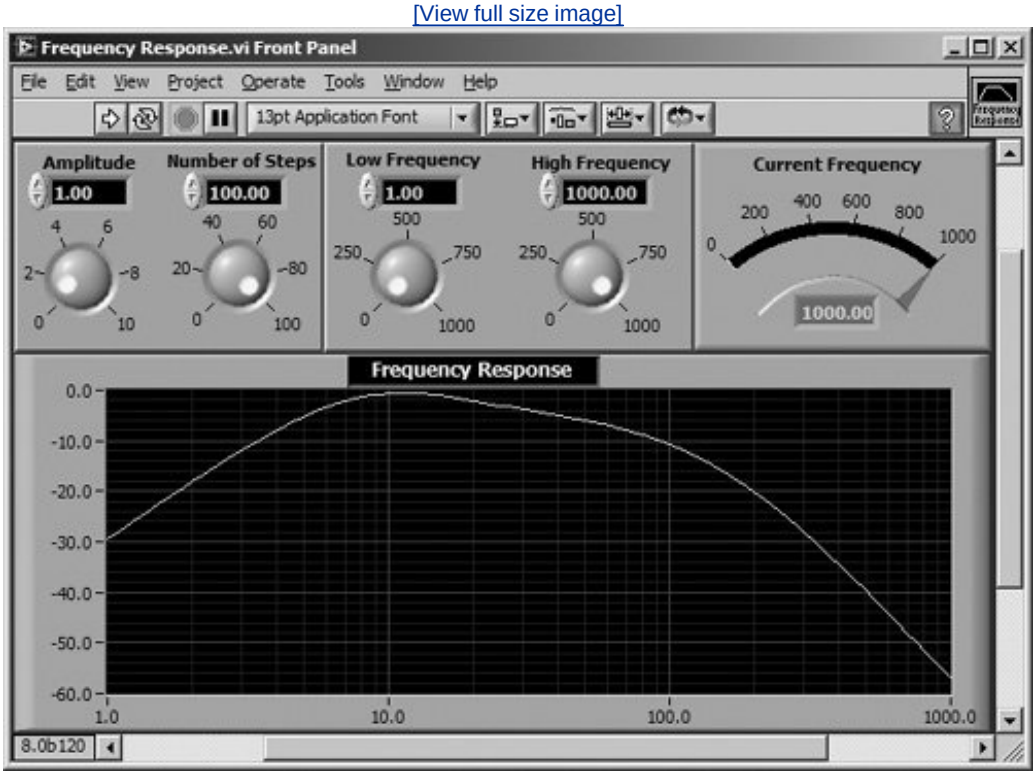
1. Launch the NI Example Finder, as described in the NI Example Finder section of this chapter.

2. With the Browse tab selected and the Browse according to: option set to Task, navigate the folder tree to "Industry Applications," and then "Instrument Input and Output." Double-click "Frequency Response.vi" to open it. (Note: You can also find this example in the LabVIEW install directory, under `examples/apps/freqresp.llb`.)

3. You will see the VI shown in Figure 1.12.

Figure 1.12. Frequency Response.vi front panel

[View full size image]



4. Run the VI by clicking on the run button. You can specify the amplitude of the input sine wave and the number of steps the VI uses to find the frequency response by changing the Amplitude control and the Number of Steps control, and then running the VI again. You can also specify the frequency sweep by inputting the upper and lower limits with the Low Frequency and High Frequency knobs. Play with these controls and observe the effect they have on the output of the "black box."


Run Button

5. Open and examine the block diagram by choosing Show Diagram from the Window menu.

6. Close the VI by selecting Close from the File menu. These exercises should give you a basic feel for LabVIEW's "G" programming environment. With the G language, you'll find writing powerful applications (and debugging them) to be a snap! Read on to learn how.

PREV NEXT

Wrap It Up!

LabVIEW is a powerful and flexible instrumentation and analysis software system. It uses a graphical programming language, sometimes referred to as "G," to create programs called *virtual instruments*, or VIs. The user interacts with the program through the *front panel*. Each front panel has an accompanying *block diagram*, which is the VI's source code. LabVIEW has many built-in functions to facilitate the programming process; components are wired together to show the flow of data within the block diagram. Stay tunedthe next chapters will teach you how to effectively use LabVIEW's many features.

Use the NI Example Finder to search for examples on the subjects you are learning. You can browse the examples by task (logical groupings) or by directory structure (how the examples are organized on disk).

When you're doing activities and viewing examples, make sure to check out the example VIs and other files, located in the `EVERYONE` directory of the CD that accompanies this book.

You will find the solutions to every activity in the upcoming chapters in the `EVERYONE` *directory on the CD that accompanies the book. We'll trust you not to cheat!*

Additional Activities

Activity 1-3: More Neat Examples

In this activity, you will look at some example programs that ship with LabVIEW.

1. From the Help menu, choose Find Examples. . . .
2. This will bring up the NI Example Finder. You can browse the tree of examples by example folder directory structure or by program task type. Double-clicking a VI will open it in LabVIEW.


Run Button

3. Run the VI by clicking on the Run button.
4. After you run an example, choose Show Diagram from the Window menu to see what the program looks like.
5. Now look through and run other VIs in the Example Finder. Try to get an idea of the LabVIEW environment and what you can do with it. Although all of the examples are extremely educational, you should investigate the examples found in the Industry Applications folder of the Task view. Feel free to browse through any VIs that strike your fancy; you can learn a lot just by watching how they work. Also feel free to modify and use these examples for your own applications (just be sure to save them to a different location so you don't overwrite the built-in examples).
6. When you're done, select Close from the File menu to close each VI. Do not save any changes you may have made.

2. Virtual Instrumentation: Hooking Your Computer Up to the Real World

Overview

Virtual instrumentation is the foundation for the modern laboratory. A virtual instrument consists of a computer, software, and modular hardware; all combined and configured to emulate the function of traditional hardware instrumentation. It's also what we call a LabVIEW program. Because their functionality is software-defined by the user, virtual instruments are extremely flexible, powerful, and cost-effective. This chapter explains how to communicate with the outside world (e.g., take measurements, "talk" to an instrument, send data to another computer) using LabVIEW. We're only giving you a very brief overview here; you can learn more about acquiring data, controlling instruments, and networking your computer with LabVIEW in the second half of this book. In this chapter, you'll also learn a little about how LabVIEW has changed over the years.

Goals

- Understand the nature of data acquisition and instrument control
- Be able to describe the components of typical DAQ or instrumentation systems
- Learn about your computer's serial, network, and USB ports
- Appreciate the usefulness of analysis functions
- Learn a little about GPIB, PXI, and VXI
- See how LabVIEW can exchange data with other computers and applications
- Learn about some of the toolkits that enhance LabVIEW's capabilities
- Learn about the LabVIEW RT, LabVIEW PDA, and LabVIEW FPGA modules

Key Terms

- Data acquisition (DAQ)
- General Purpose Interface Bus (GPIB)
- Institute of Electrical and Electronic Engineers (IEEE) 488 standard
- Serial port
- PXI
- VXI
- Internet capabilities
- Networking
- Shared library and dynamic link library (DLL)
- Code interface node (CIN)
- ActiveX and .NET
- Toolkit

Using LabVIEW in the Real World

Although LabVIEW is a very powerful simulation tool, it is most often used to gather data from an external source, and it contains many VIs built especially for this purpose. For example, LabVIEW can command plug-in data acquisition, or DAQ, devices to acquire or generate analog and digital signals. You might use DAQ devices and LabVIEW to monitor a temperature, send signals to an external system, or determine the frequency of an unknown signal. LabVIEW also facilitates data transfer over the General Purpose Interface Bus (GPIB), or through your computer's built-in USB, Ethernet, Firewire (also known as IEEE 1394), or serial port. GPIB is frequently used to communicate with oscilloscopes, scanners, and multimeters, and to drive instruments from remote locations. LabVIEW software can also control sophisticated VXI hardware instrumentation systems, Ethernet, or USB-based instruments. Once you have acquired or received your data, you can use LabVIEW's many analysis VIs to process and manipulate it.

Often you will find it useful to share data with other applications or computers in addition to an instrument. LabVIEW has built-in functions that simplify this process, supporting several networking protocols, external calls to existing code or dynamic link libraries (DLLs), and ActiveX automation.

We'll spend the rest of this chapter talking about some of the tasks LabVIEW was designed to accomplish. But, before we start talking about exactly what LabVIEW can do, let's take a moment to reflect on how LabVIEW got to be the powerful tool that it is today. This next section gives a brief history lesson, showing the LabVIEW timeline and the introduction of its many features and capabilities.

The Evolution of LabVIEW

In 1983, National Instruments began to search for a way to minimize the time needed to program instrumentation systems. Through this effort, the LabVIEW virtual instrument concept evolvedintuitive front panel user interfaces combined with an innovative block diagram programming methodology to produce an efficient, software-based graphical instrumentation system.

LabVIEW version 1 was released in 1986 on the Macintosh only. Although the Mac was not widely used for measurement and instrumentation applications, its graphical nature best accommodated the LabVIEW technology until the more common operating systems could support it.

By 1990, National Instruments had completely rewritten LabVIEW, combining new software technology with years of customer feedback. More importantly, LabVIEW 2 featured a compiler that made execution speeds of VIs comparable with programs created in the C programming language. The United States Patent Office issued several patents recognizing the innovative LabVIEW technology.

As other graphical operating systems appeared, National Instruments ported the now mature LabVIEW technology to the other platforms: PCs and workstations. In 1992, they introduced LabVIEW for Windows and LabVIEW for Sun based on the new portable architecture.

LabVIEW 3 arrived in 1993 for Macintosh, Windows, and Sun operating systems. LabVIEW 3 programs written on one platform could run on another. This multiplatform compatibility gave users the opportunity to choose the development platform while ensuring that they could run their VIs on other platforms (consider that this was a couple of years before Java was introduced). In 1994, the list of LabVIEW-supported platforms grew to include Windows NT, Power Macs, and HP workstations. 1995 brought about an adaptation to Windows 95.

LabVIEW 4, released in 1996, featured a more customizable development environment so users could create their own workspace to match their industry, experience level, and development habits. In addition, LabVIEW 4 added high-powered editing and debugging tools for advanced instrumentation systems, as well as OLE-based connectivity and distributed execution tools.

LabVIEW 5 and 5.1 (in 1999) continued to improve on the development tool by introducing a built-in web server, a dynamic programming and control framework (VI Server), integration with ActiveX, and easy sharing of data over the Internet with a protocol called DataSocket.[1] The *undo* feature taken for granted in most programs finally was implemented.

[1] The DataSocket protocol has been succeeded by the NI Publish and Subscribe Protocol (NI-PSP). See Chapter 16 for more information about NI-PSP and the DataSocket VIs.

In 2000, LabVIEW 6 (sometimes called 6i) introduced support for the Linux open source operating system. It also introduced a new suite of 3-D controls; appropriately at a time when the computing industry was discovering that style did matter (spearheaded by the introduction of Apple's iMac and G4 cubes). LabVIEW 6 does a very impressive job of providing both an easy and intuitive programming interface (especially for non-programmers), as well as supporting a slew of advanced programming techniques, such as object-oriented development, multithreading, distributed computing, and much more. Don't let the graphical nature of LabVIEW fool you: LabVIEW is a tool

that can easily rival C++ or Visual Basic as a development toolwith the benefit, as thousands have discovered, that it's much more fun!

In 2001, LabVIEW 6.1 introduced event-oriented programming, remote web control of LabVIEW, Remote Front Panels, VISA support for communication with infrared devices (IrDA), and other improvements.

Also in 2001, LabVIEW Real-Time (LabVIEW RT) was introduced, allowing VIs developed in LabVIEW to be downloaded to the RT Engine of National Instruments RT Series devices and run in real time. (We will discuss LabVIEW RT more, later in this chapter.)

In 2003, LabVIEW 7.0 (sometimes called 7 Express) introduced several new features for both beginning and advanced users. Most notable are the *Express Technologies:* a framework of tools designed to get beginning LabVIEW users up and running quickly by providing easily configurable, ready-to-use subVIs and functions. For the advanced user, LabVIEW 7.0 extended the functionality of the event structure to include user-defined events as well as a dynamic event registration frameworkno longer was the event structure bound only to the events on the Front Panel of its containing VI. Other new additions included the Tree Control and the SubPanel, for creating more flexible and powerful user interfaces. And, several editor enhancements like the snap-to-grid, resizing tool, grab-handles, as well as many others made using LabVIEW all the more enjoyable.

Also in 2003, the LabVIEW PDA and LabVIEW FPGA modules were introduced. LabVIEW PDA module allowed creating LabVIEW programs that could run on PalmOS and PocketPC. LabVIEW FPGA allowed creating LabVIEW programs that could run on Field Programmable Gate Arrays (FGPA) of National Instrument's FGPA devices. (We will discuss LabVIEW PDA and LabVIEW FPGA more, later in this chapter.)

In 2004, LabVIEW 7.1 added VISA support for communication with Bluetooth devices, native Radio Buttons control, a Navigation Window, and many other useful features, including the evolution of *Express Technologies,* such as the Timed Loop that provides precision timing on real-time and FPGA targets as well as synchronization capabilities.

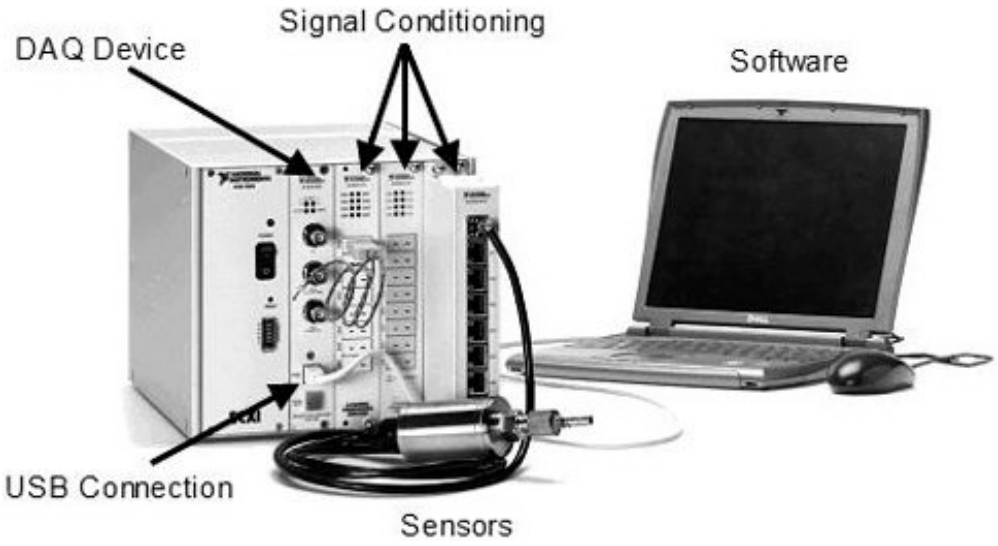
In 2005, LabVIEW 8 introduced the Project Explorer, an IDE-style workspace that allows developers to manage the development of a Virtual Instrumentation system. A LabVIEW project can contain VIs, hardware resources, and configurations, as well as build and deployment rules. LabVIEW 8 also added support for Project Library components, niceties such as right-click menus and drag-and-drop, and custom controls with edit-time behavior.

PREV NEXT

What Is Data Acquisition?

Data acquisition, or DAQ for short, is simply the process of measuring a real-world signal, such as a voltage, and bringing that information into the computer for processing, analysis, storage, or other data manipulation. Figure 2.1 shows the components of a DAQ system. Physical phenomena (which are *not* shown in Figure 2.1) represent the real-world signals you are trying to measure, such as speed, temperature, humidity, pressure, flow, pH, start-stop, radioactivity, light intensity, and so on. You use sensors (sometimes also called transducers) to evaluate the physical phenomena and produce electrical signals proportionately. For example, thermocouples, a type of sensor, convert temperature into a voltage that an A/D (analog to digital) converter can measure. Other examples of sensors include strain gauges, flowmeters, and pressure transducers, which measure displacement in a material due to stress, rate of flow, and pressure, respectively. In each case, the electrical signal produced by the sensor is directly related to the phenomenon it monitors.

Figure 2.1. DAQ system

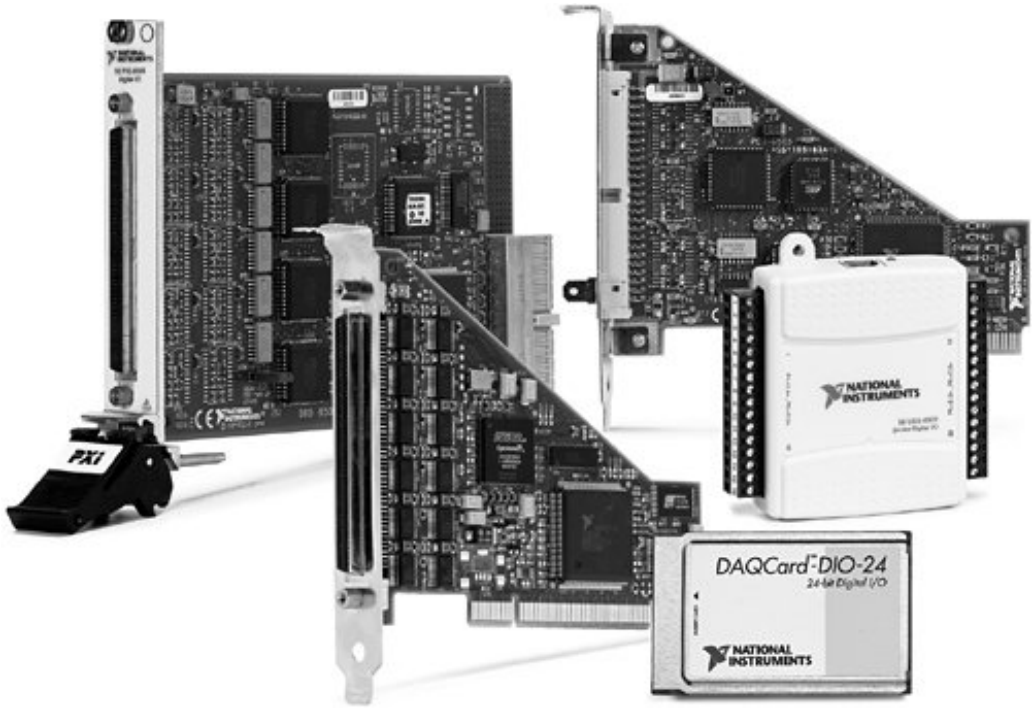


LabVIEW can command DAQ devices to read analog input signals (A/D conversion), generate analog output signals (D/A conversion), read and write digital signals, and manipulate the on-board counters for frequency measurement, pulse generation, quadrature encoder measurements, and so on, to interface with the transducers. In the case of analog input, the voltage data from the sensor goes into the plug-in DAQ devices in the computer, which sends the data into computer memory for storage, processing, or other manipulation.

Signal conditioning modules "condition" the electrical signals generated by transducers so that they are in a form that the DAQ devices can accept. For example, you would want to isolate a high-voltage

input such as 120 VAC, lest you fry both your board and your computera costly mistake! Signal conditioning modules can apply to many different types of conditioning: amplification, linearization, filtering, isolation, and so on. Not all applications will require signal conditioning, but many do, and you should pay attention to your specifications to avoid a potential disaster. In addition, information loss can be even worse than equipment loss! Noise, nonlinearity, overload, aliasing, etc. can hopelessly corrupt your data, and LabVIEW will not save you. Signal conditioning is often not optionalit's best to check before you start.

Figure 2.2. Many types of DAQ devices are available from NI.



To acquire data in your lab using the virtual instrumentation approach, you will need a DAQ device, a computer configured with LabVIEW and DAQ driver software, and some method of connecting your transducer signal to the DAQ device, such as a connector block, breadboard, cable, or wire. You may also need signal conditioning equipment, depending on the specifications of your application.

For example, if you wanted to measure a temperature, you would need to wire the temperature sensor to an analog input channel on the DAQ device in your computer (often via signal conditioning equipment, depending on the sensor). Then use LabVIEW's DAQ VIs to read the channel on the board, display the temperature on the screen, record it in a data file, and analyze it any way you need to.

The built-in LabVIEW data acquisition VIs only work with National Instruments' DAQ devices. If you are using a board from another vendor, you will have to get a driver from them (if they have one), or you will have to write your own driver code and call it from LabVIEW using code interface nodes or dynamic link libraries.

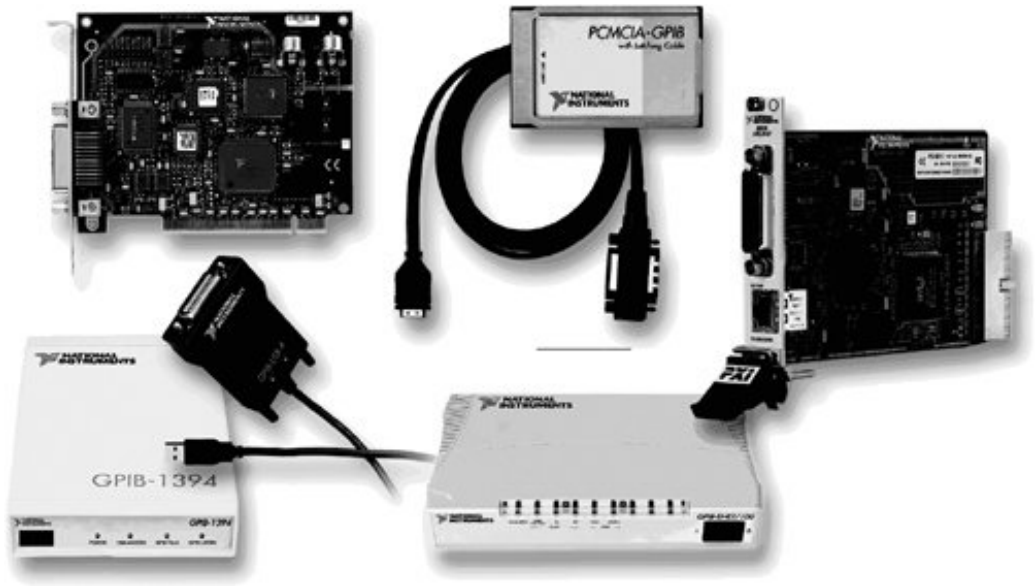
PREV NEXT

What Is GPIB?

Hewlett Packard developed the General Purpose Interface Bus, or GPIB, in the late 1960s to facilitate communication between computers and instruments. A bus is simply the means by which computers and instruments transfer data, and GPIB provided a much-needed specification and protocol to govern this communication. The Institute of Electrical and Electronic Engineers (IEEE) standardized GPIB in 1975, and it became known as the IEEE 488 standard (GPIB = IEEE 488). GPIB's original purpose was to provide computer control of test and measurement instruments. However, its use has expanded beyond these applications into other areas, such as computer-to-computer communication and control of multimeters, scanners, and oscilloscopes.

GPIB is a parallel bus that many instruments use for communication. GPIB sends data in bytes (one byte = eight bits), and the messages transferred are frequently encoded as ASCII character strings. Your computer can only perform GPIB communication if it has a GPIB board (or external GPIB box), such as those shown in Figure 2.3, and the proper drivers installed.

Figure 2.3. GPIB boards from NI

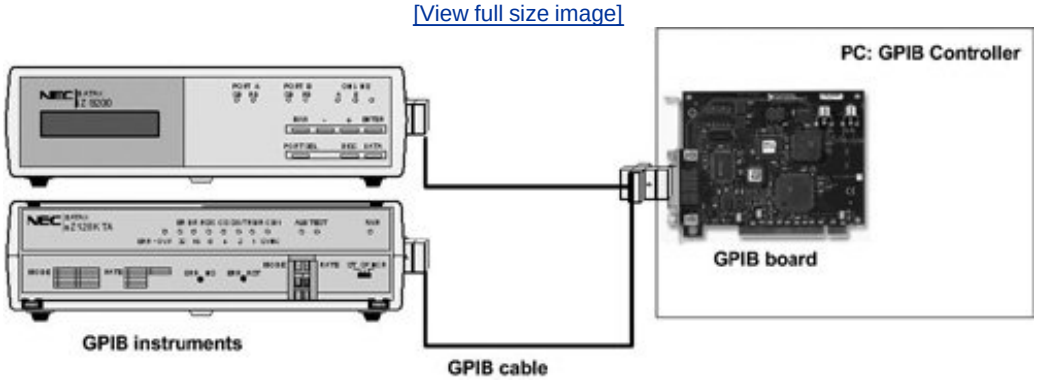


You can have many instruments and computers connected to the same GPIB bus. Every device, including the computer interface board, must have a unique GPIB address between 0 and 30, so that the data source and destinations can be specified by this number. Address 0 is normally assigned to the GPIB interface board. Instruments connected to the bus can use addresses 1 through 30. The GPIB has one Controller, usually your computer, that controls the bus management functions. To transfer instrument commands and data on the bus, the Controller addresses one Talker and one or

more Listeners. The data strings are then sent across the bus from the Talker to the Listener(s). The LabVIEW GPIB VIs automatically handle the addressing and most other bus management functions, saving you the hassle of low-level programming. The following illustration shows a typical GPIB system (see Figure 2.4).

Figure 2.4. Typical GPIB system containing one or more GPIB enabled instruments and a GPIB controller board

[View full size image]



Although using GPIB is one way to bring data into a computer, it is fundamentally different from performing data acquisition, even though both use boards that plug into the computer. Using a special protocol, GPIB talks to another computer or instrument to bring in data acquired by that device, while data acquisition involves connecting a signal directly up to a computer's DAQ device.

To use GPIB as part of your virtual instrumentation system, you need a GPIB board or external box, a GPIB cable, LabVIEW and a computer, and an IEEE 488-compatible instrument with which to communicate (or another computer containing a GPIB board). You also need to install the GPIB driver software on your computer as well, according to the directions that accompany LabVIEW or the board.

LabVIEW's GPIB VIs communicate with National Instruments' GPIB boards, but not those from other manufacturers. If you have another vendor's board, you can either get driver software from them (if it's available) or write your own driver code and integrate it into LabVIEW. As with DAQ drivers, it's not an easy thing to do!

We'll talk more about DAQ and GPIB in Chapters 10, "Signal Measurement and Generation: Data Acquisition," 11, "Data Acquisition in LabVIEW," and 12, "Instrument Control in LabVIEW."

Communication Using the Serial Port

Serial communication is another popular means of transmitting data between a computer and a peripheral device such as a programmable instrument (or even another computer).

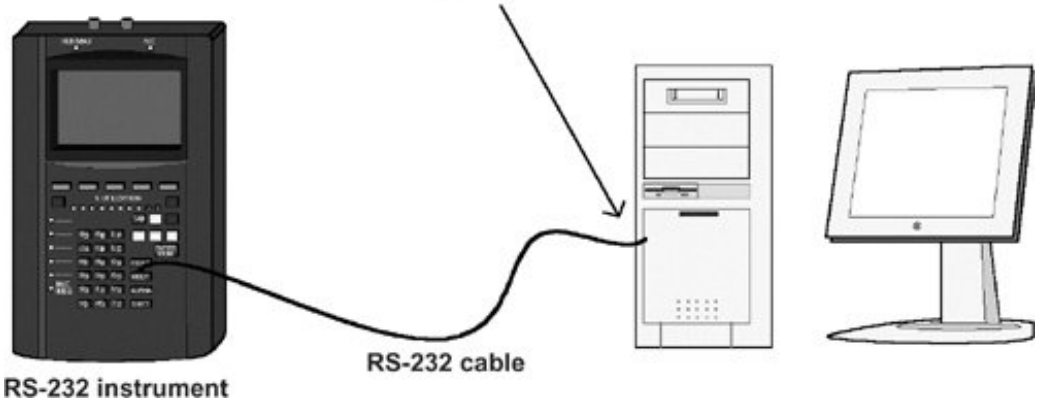
LabVIEW can perform serial communication (either RS-232, RS-422, or RS-485 standards) using built-in or externally attached (for example, USB serial adaptors) serial ports on your computer.[2] Serial communication uses a transmitter to send data one bit at a time over a single communication line to a receiver. You can use this method when data transfer rates are low, or when you must transfer data over long distances. The old-fashioned serial communication protocol, RS-232, is slower and less reliable than the GPIB, but you do not need a board in your computer to do it, your instrument does not need to conform to the IEEE 488 standard, and many devices still work with RS-232.

[2] Built-in serial ports on a computer are almost always RS-232. And because RS-232 is one of the most common types of serial communication, sometimes RS-232 ports are simply referred to as "serial ports."

Figure 2.5 shows a typical serial communication system.

Figure 2.5. Typical (RS-232) serial system containing one RS-232 enabled instrument connected to a computer via its serial port

[View full size image]



Serial communication is handy because most PCs have one or two RS-232 serial ports built inyou can send and receive data without buying any special hardware. Some newer computers do not have a built-in serial port, but it is easy to buy a USB to RS-232 serial adaptor for about the cost of a USB mouse. Although most computers also now have USB (universal serial bus) ports built-in, USB is a more complex protocol that is oriented at computer peripherals, rather than communication with scientific instruments. Serial communication (RS-232, RS-422, or RS-485) is old compared to USB,

but is still widely used for many industrial devices.

Many GPIB instruments also have built-in serial ports. However, unlike GPIB, an RS-232 serial port can communicate with only one device, which can be limiting for some applications.[3] Serial port communication is also painstakingly slow and has no built-in error-checking capabilities. However, serial communication has its uses (it is certainly economical), and the LabVIEW Serial library contains ready-to-use functions for serial port operations. If you have a cable and a device to "talk" to, you are all set to try out serial communication!

[3] RS-422 and RS-485 are commonly called "multi-drop serial" and can facilitate communication between multiple devices on a single bus. RS-422 and RS-485 are also less susceptible to noise and allow longer cable lengths, which are reasons that they are commonly preferred (over RS-232) for industrial applications.

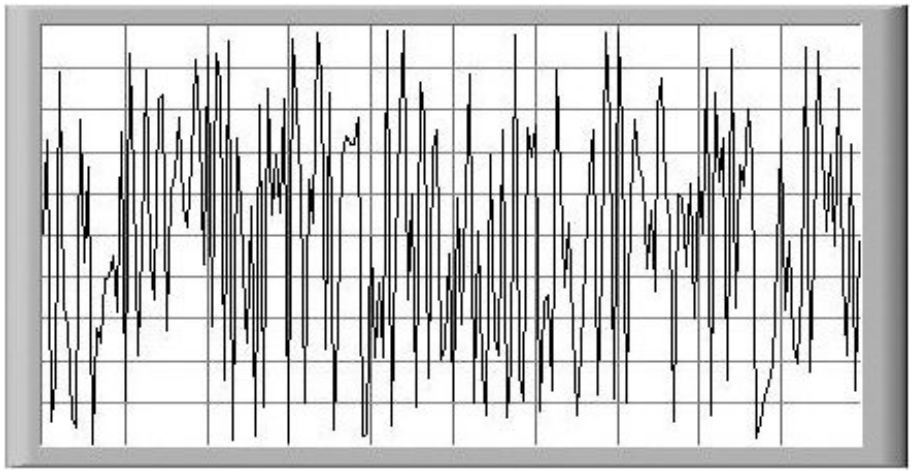
PREV NEXT

Real-World Applications: Why We Analyze

Once you get data into your computer, you may want to process your data somehow. A few of the many possible analysis applications for LabVIEW include biomedical data processing, speech synthesis and recognition, and digital audio and image processing.

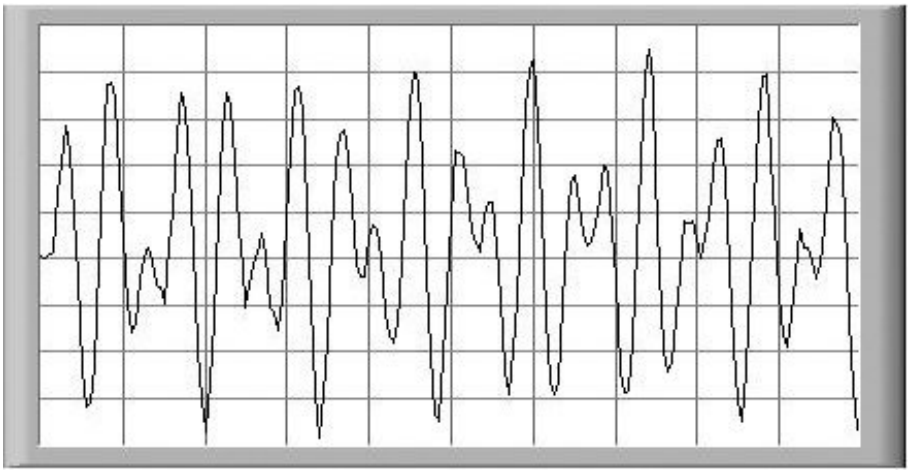
The importance of integrating analysis libraries into laboratory stations is obvious: The raw data collected from your DAQ device or GPIB instrument does not always immediately convey useful information. Often you must transform the signal, remove noise, correct for data corrupted by faulty equipment, or compensate for environmental effects such as temperature and humidity. Figure 2.6 shows data containing noise that is obscuring our signal of interest.

Figure 2.6. Data containing noise



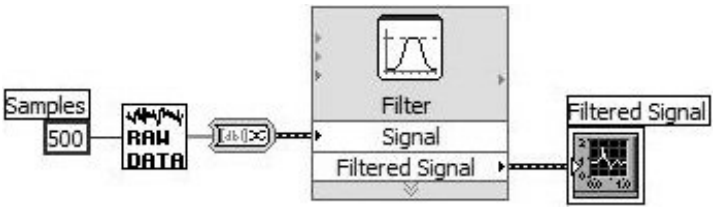
By analyzing and processing the digital data, you can extract the useful information from the noise and present it in a form more comprehensible than the raw data. The processed data looks more like what is displayed in Figure 2.7.

Figure 2.7. Data that has been processed to remove noise



The LabVIEW block diagram programming method and the extensive set of LabVIEW analysis VIs simplify the development of analysis applications. The following sample block diagram illustrates the LabVIEW programming concept (see Figure 2.8).

Figure 2.8. Block diagram showing how you might process raw data and graph the results in LabVIEW



Because the LabVIEW analysis functions give you popular data analysis techniques in discrete VIs, you can wire them together, as shown in the previous picture, to analyze data. Instead of worrying about implementation details for analysis routines as you do in most programming languages, you can concentrate on solving your data analysis problems. LabVIEW's analysis VIs are powerful enough for experts to build sophisticated analysis applications using digital signal processing (DSP), digital filters, statistics, or numerical analysis. At the same time, they are simple enough for novices to perform sophisticated calculations.

The LabVIEW analysis VIs efficiently process blocks of information represented in digital form. They cover the following major processing areas:

- Signal and noise generation
- Spectral analysis (FFT, power spectrum, etc.)
- Pulse characterization (timing, transition, levels, etc.)

- Amplitude/level measurements (DC/RMS, peak, etc.)
- Distortion measurements (SINAD, THD, etc.)
- Signal monitoring (limit mask, triggering, etc.)
- Curve fitting/optimization
- Interpolation/extrapolation
- BLAS/LAPACK-based linear algebra
- Point-by-point analysis
- Probability and statistics
- Transforms (Fourier, Hilbert, etc.)
- Frequency and impulse response
- Peak/level detection
- Digital filtering
- Signal resample/align
- Windowing
- Ordinary differential equations
- Integration and differentiation
- Polynomial functions/root solving
- Elementary and special functions

PREV NEXT

A Little Bit About PXI and VXI

Two other hardware platforms you should be familiar with are PXI and VXI.

PXI, an acronym for *compactPCI eXtensions for Instrumentation,* defines a modular hardware platform based on the Intel x86 processor (PC architecture) and the CompactPCI bus (a version of the PCI bus). A typical configuration involves a PXI chassis (such as the one shown in Figure 2.9), which holds its own PC computer (called a *controller*) running Microsoft Windows and slots for all types of measurement modules: analog input, imaging, motion control, sound, relays, GPIB and VXI interfaces, and more. The compact, ruggedized, and expandable system make it an attractive platform for many applications. Figure 2.10 shows a PXI system in a real-world application. In addition to a standard PXI configuration running Microsoft Windows on the controller, you can use the real-time version of LabVIEW (LabVIEW RT) on the controller for a more robust system, or even run Linux (support for NI hardware on Linux is steadily improving).

Figure 2.9. PXI chassis with controller and various IO cards installed

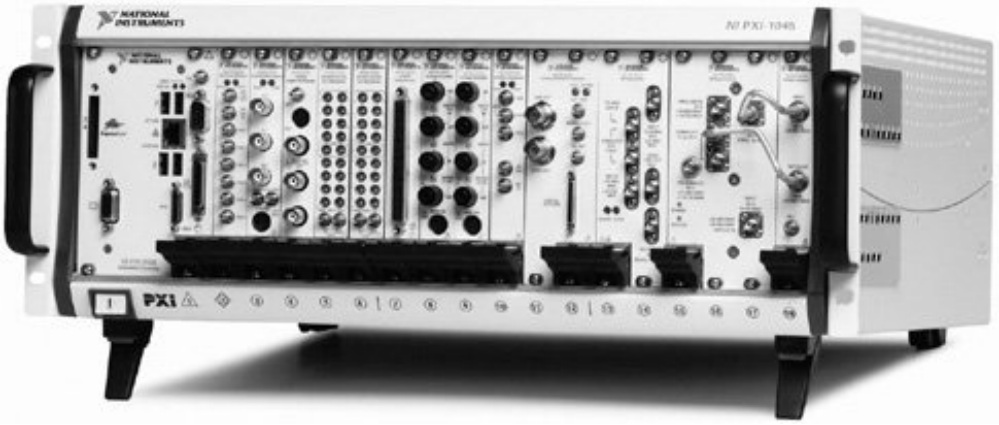


Figure 2.10. PXI system

VXI, an acronym for *VME eXtensions for Instrumentation,* is another instrumentation standard for instrument-on-a-card systems. First introduced in 1987 and based on the VMEbus (IEEE 1014) standard, the VXIbus is a higher-end and usually more costly system than PXI. VXI consists of a mainframe chassis with slots holding modular instruments on plug-in boards. A variety of instrument and mainframe sizes is available from numerous vendors, and you can also use VME modules in VXI systems. VXI has a wide array of uses in traditional test and measurement and ATE (automated test equipment) applications. VXI is also popular in data acquisition and analysis for research and industrial control applications that require high number of channels (hundreds or thousands).

VXIplug&play is a name used in conjunction with VXI products that have additional standardized features beyond the scope of the baseline specifications. VXIplug&play-compatible instruments include standardized software, which provides soft front panels, instrument drivers, and installation routines to take full advantage of instrument capabilities and make your programming task as easy as possible. LabVIEW software for VXI is fully compatible with VXIplug&play specifications.

PREV NEXT

Connectivity

In some applications, you will want to share data with other programs, perhaps locally or across your local network. In many cases, you may want to share the data over the Internet and allow other people to view or control your system over the Web.

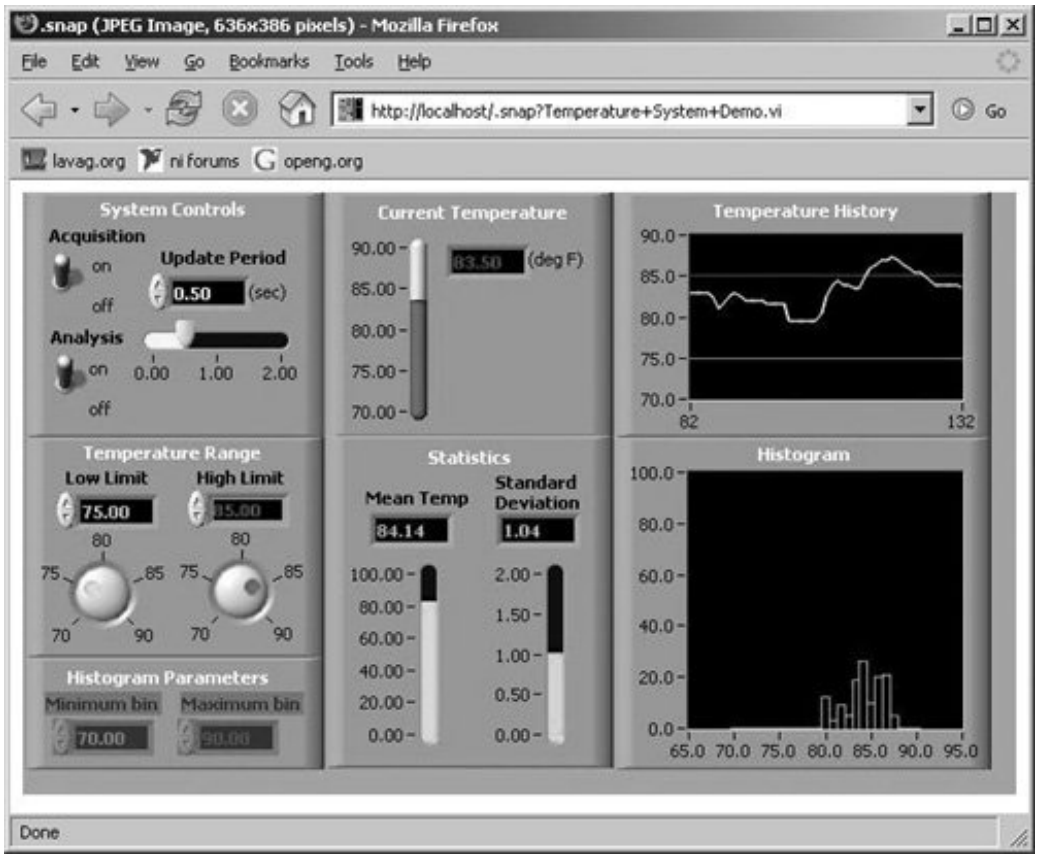
LabVIEW has built-in features (such as a web server, web publishing tool, email VIs, and network variables) and functions that simplify this process. These VIs facilitate communication over a network or over the Internet. LabVIEW can use the NI Publish and Subscribe Protocol (NI-PSP), which we'll learn more about in Chapter 16, "Connectivity in LabVIEW," to share data across networks, call and create dynamic link libraries (DLLs) or external code, and support ActiveX automation and .NET assemblies. Using additional add-on modules and toolkits, LabVIEW can also communicate with most SQL (structured query language) databases, such as MySQL, PostgreSQL, Oracle, SQL Server, and Access. Network variables can be defined in a LabVIEW project and easily shared across an entire distributed measurement system.

Internet Connectivity

LabVIEW has several built-in features that make it very easy to share your VIs and data over the Internet. Using LabVIEW's web server and Remote Panels, you can allow people to remotely view and control your VIs (without any additional programming), as shown in Figure 2.11.

Figure 2.11. A VI's front panel image viewed in Mozilla Firefox, generated by LabVIEW's web server

[View full size image]



With the *Enterprise Connectivity* *Toolkit*, you can also use LabVIEW to ftp and telnet into remote systems, as well as provide enhanced web-serving capabilities. Emailing capabilities are built into the LabVIEW Professional version.

Networking

For our purposes, *networking* refers to communication between multiple processes that usually (but not necessarily) run on separate computers. This communication can occur on a closed, local area network (LAN), or over the Internet. One main use for networking in software applications is to allow one or more application to use the services of another application. In addition to the web publishing features of LabVIEW, you can use some of the networking functionality to communicate with other software or another LabVIEW program.

For communication between processes to work, the processes must use a common communications language, referred to as a protocol. LabVIEW supports the following protocols:

- NI Publish and Subscribe Protocol (NI-PSP) a proprietary protocol from National Instruments for sharing instrumentation data. It has the advantage that it is very easy to use. *Shared Variables* (discussed in Chapter 16) use this technology under the hood.
- TCP / IP the basic protocol of most networks, including the Internet.

- UDP a basic protocol, similar to TCP/IP but without acknowledged data delivery.

This book will talk more about Internet capabilities and networking in Chapter 16.

ActiveX and .NET

ActiveX is a technology from Microsoft that defines a component-based architecture for building applications that can communicate with each other. ActiveX builds upon previous technologies such as OLE. With ActiveX, one application can share a piece of code (a *component*) with a completely different application. For example, because Microsoft Word is an ActiveX component, you can control and embed a Word document inside another ActiveX-capable application, such as LabVIEW VI. LabVIEW supports ActiveX automation and can contain ActiveX components.

The .NET framework is a newer technology from Microsoft designed to simplify application development in the highly-distributed environment of the Internet. In LabVIEW, you can easily connect to this new .NET technology as a .NET client. You can create instances of .NET classes and invoke methods to set and obtain properties on them. In this way, using .NET in LabVIEW is similar to using ActiveX automation VIs.

If you don't understand what we're talking about, don't worry. (ActiveX and .NET are fairly complicated advanced topics. They're discussed in more detail in Chapter 16.)

Shared Libraries, DLLs, and CINs

For increased flexibility, LabVIEW can both call and create external code routines (for example, code written in C++) as *shared libraries* and integrate these routines into program execution. In case you were wondering, a shared library is a library of shared functions that an application can link to at runtime, instead of at compile time. On Windows, shared libraries are called *dynamic link libraries (DLLs);* on Mac OS X, they are called *frameworks;* and on Linux, they are called shared objects (or just shared libraries).

You can use the Call Library Function in LabVIEW to call a shared library. You can also tell LabVIEW to compile its VIs as a shared library that other types of code (for example, C++) can use. Shared libraries on other (non-Windows) platforms do not have a `.dll` file extensionthere is a listing of platform-specific shared library file extensions in Table 2.1.

Table 2.1. Shared Library File Extensions

Operating System	*Shared Library File Extension*
Windows	`.dll`
Mac OS X	`.framework`
UNIX	`.so`

In addition to calling shared libraries, LabVIEW can also call external code using a special block diagram structure called a *code interface node (CIN)* to link conventional, text-based code to a VI. LabVIEW calls the executable code when the node executes, passing input data from the block diagram to the executable code, and returning data from the executable code to the block diagram. CINs are sometimes used when the external code doesn't fit neatly into a shared library structure. It is also important to note that CINs are statically linked into the VI. The CIN cannot be changed without editing the VI. (This is not true for shared libraries, which are only linked at runtime.) Also, VIs that use CINs are not cross-platform compatible.

Most applications never require the use of a CIN or DLL. Although the LabVIEW compiler can usually generate code that is fast enough for most tasks, CINs and DLLs are useful for some tasks that are time-critical, require a great deal of data manipulation, or that you have already written specific code for. They are also useful for tasks that you can't perform directly from the diagram, such as calling operating system routines for which LabVIEW functions do not exist. Also, many third-party (non-LabVIEW) software developers package their software as shared libraries. For example, the VXI plug&play compatible drivers mentioned in the section, "A Little Bit About PXI and VXI," are actually shared libraries (DLLs) with LabVIEW VIs that call the DLL functions.

Other Communication Mechanisms

In addition to these network protocols, LabVIEW provides support for some older and lesser-used protocols and application communication platforms, such as *DDE* (Windows), as well as named pipes

LabVIEW Add-on Toolkits

You can use the following special add-on toolkits with LabVIEW to increase your flexibility and capabilities. For more information about them, see Appendix B, "Add-on Toolkits for LabVIEW." Some of the more common toolkits are the following:

- Application Builder
- Enterprise Connectivity Toolkit
- Internet Toolkit
- Database Connectivity Toolkit
- SPC (Statistical Process Control) Toolkit
- Sound & Vibration Analysis Toolkit
- OpenG Add-ons for LabVIEW
- LabSQL, Free Open Source Database Toolkit
- LabVIEW Vision Development Module
- VI Analyzer Toolkit

Some toolkits are sold by National Instruments; others by third-party companies, such as National Instruments Alliance Members or open source software developers that have made add-ons to LabVIEW that do all sorts of things. If you have a specific task and you want to know if someone has already done it, we suggest posting to one of the many mailing lists and discussion forumsfor example, the Info-LabVIEW mailing list, NI discussion forums, LabVIEW Advanced Virtual Architects (LAVA) discussion forums, or OpenG.org discussion forums (see Appendix E, "Resources for LabVIEW," for details).

LabVIEW Real-Time, FPGA, PDA, and Embedded

There are a few special add-on modules for LabVIEW that we should mention. These are the LabVIEW Real-Time, LabVIEW FPGA, LabVIEW PDA, and LabVIEW Embedded modules, which allow you to run your LabVIEW VIs on other execution targets.

The LabVIEW Real-Time module is a hardware and software combination that allows you to take portions of your LabVIEW code, and download them to be executed on a separate controller board with its own real-time operating system. This means that you can guarantee that certain pieces of your LabVIEW program will keep running with precision, even if your user interface host machine crashes and your computer screeches to a halt.

The LabVIEW FPGA and LabVIEW PDA modules allow you to target your LabVIEW programs to run on a Field Programmable Gate Array (FPGA) or Personal Digital Assistant (PDA), respectively. The LabVIEW FPGA module provides the capability to leverage the inherently parallel nature of data-flow programming with the inherently parallel programmable logic device, the FPGA chip. The LabVIEW PDA module allows developers to deploy LabVIEW applications on hand-held devices such as those running Palm OS and Pocket PC, for creating hand-held, portable, networked data acquisition systems.

The LabVIEW Embedded Module allows you to compile your LabVIEW VIs and run them on *any* (yes, any!) 32-bit microprocessor platform by integrating LabVIEW with third-party toolchains. This includes the GNU C++ (gcc) compiler, eCos, Wind River Tornado/VxWorks, and many other platformsand you can create your own toolchain support layer, so the sky is the limit.

As we move into the future, expect to be able to run your LabVIEW VIs. . . well. . . everywhere! "LabVIEW Everywhere" is the philosophy that you should be able to develop LabVIEW virtual instruments and deploy them onto just about any hardware, even reconfiguring the hardware to perform in ways that the hardware designers had not anticipated. National Instruments is committed to this philosophy and will undoubtedly provide more and better tools and modules for running "LabVIEW Everywhere."

Wrap It Up!

LabVIEW's built-in functions facilitate hardware communication with external devices so you do not have to write involved programs. LabVIEW virtual instruments can work with several types of hardware to gather or exchange data: plug-in DAQ devices, GPIB controllers, serial ports (whether it's built-in or a USB-to-serial adaptor), or PXI and VXI hardware. You can use National Instruments DAQ devices, managed by LabVIEW, to read and generate analog input, analog output, and digital signals and also to perform counter/timer operations. LabVIEW can also control communication over the GPIB bus (assuming you have a GPIB controller) or command a PXI or VXI instrumentation system. If you don't have any special hardware, LabVIEW can communicate with other devices through your computer's serial port.

LabVIEW analysis VIs make it easy for you to process and manipulate data once you have brought it into your computer. Rather than working through tricky algorithms by hand or trying to write your own low-level code, you can simply access the built-in LabVIEW functions that suit your needs.

You can use LabVIEW's built-in Internet capabilities to publish images of your VIs to the web, as well as communicate with other programs and computers linked by a network using protocols such as NI-PSP or TCP/IP. LabVIEW supports ActiveX and .NET to communicate with other programs, and can both call and create shared libraries (such as DLLs, on Windows).

If you want to expand LabVIEW's considerable functionality, you can buy add-on toolkits to accomplish specific tasks. Toolkits are available to build standalone LabVIEW applications, enhance its Internet capabilities, design and analyze certain types of signals, perform statistical process and PID control, communicate with an SQL database, and much more. You can also download free add-ons from OpenG.org, an open source community that collaboratively develops LabVIEW add-ons.

The next chapters will teach you the fundamentals of LabVIEW programming, so get ready to writeer, that is, *draw*some code!

3. The LabVIEW Environment

Overview

In this chapter, you will investigate the LabVIEW VI and learn how its three partsthe front panel, block diagram, and icon/connectorwork together. When all three main components are properly developed, you have a VI that can stand alone or be used as a subVI in another program. You will also learn about the LabVIEW environment: the LabVIEW Project Explorer, pull-down and pop-up menus, floating palettes and subpalettes, the Toolbar, and how to get help. To finish up, we will discuss the power of subVIs and why you should use them.

Goals

- Understand and practice using the front panel, block diagram, and icon/connector
- Learn the difference between controls and indicators
- Be able to recognize the difference between the block diagram terminals of controls and indicators
- Understand the principle of dataflow programming
- Learn about the LabVIEW Project Explorer and how to manage your project files
- Become familiar with LabVIEW menus, both pop-up and pull-down
- Learn about the capabilities and uses of the Toolbar, Tools palette, Controls palette, Functions palette, and subpalettes
- Learn why the Help windows can be your most valuable ally
- Understand what a subVI is and why it's useful
- Work through the activities to get a feel for how LabVIEW works

Key Terms

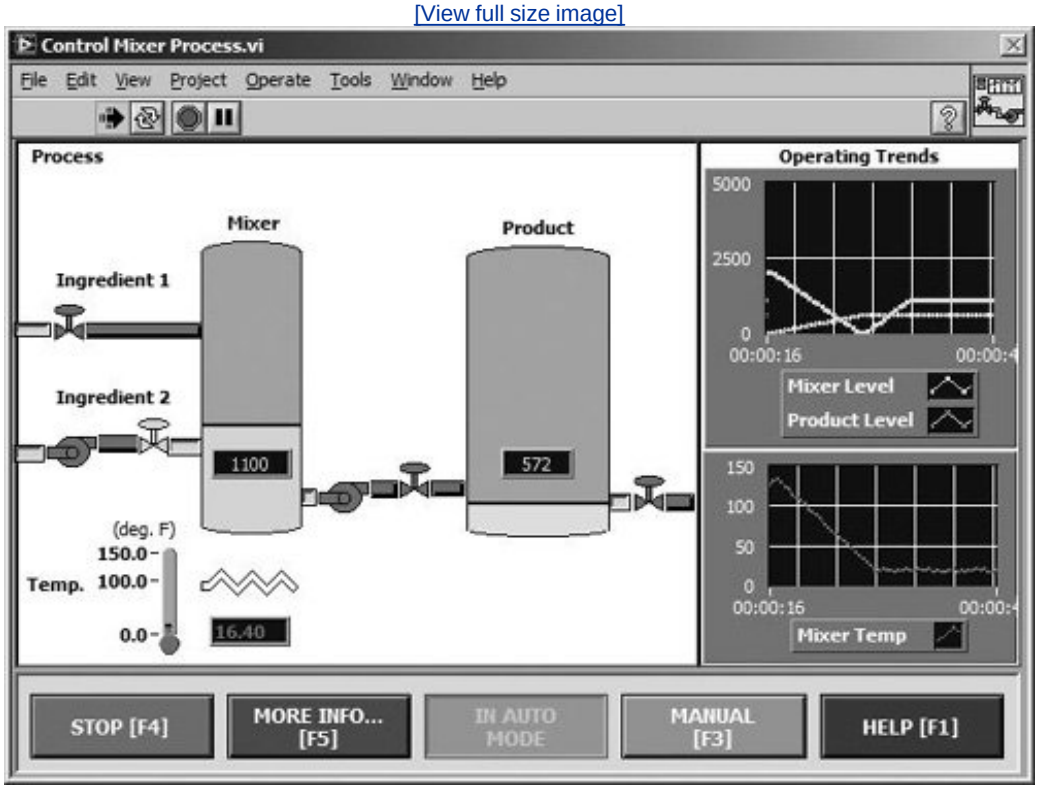
- Control
- Indicator
- Wire
- SubVI
- Express VI
- Terminal
- Node
- Dataflow
- Pop-up menus
- Toolbar
- Palette
- Subpalette
- Help window

Front Panels

Simply put, the *front panel* is the window through which the user interacts with the program. When you run a VI, you must have the front panel open so that you can input data to the executing program. You will also find the front panel indispensable because that's where you see your program's output. Figure 3.1 shows an example of a LabVIEW front panel.

Figure 3.1. LabVIEW front panel

[View full size image]



Controls and Indicators

The front panel is primarily a combination of controls and indicators. Controls simulate typical input objects you might find on a conventional instrument, such as knobs and switches. Controls allow the user to input values; they supply data to the block diagram of the VI. Indicators show output values produced by the program. Consider this simple way to think about controls and indicators:

Controls = Inputs from the User = Source of Data

Indicators = Outputs to the User = Destinations or "Sinks" for Data

They are generally not interchangeable, so make sure you understand the difference.

You "drop" controls and indicators onto the front panel by selecting them from a subpalette of the floating Controls palette window and placing them in a desired spot. Once an object is on the front panel, you can easily adjust its size, shape, position, color, and other properties.

You will later learn (when we discuss local variables, in Chapter 13, "Advanced LabVIEW Structures and Functions," and the Value property accessible using control references, in Chapter 15, "Advanced LabVIEW Features") that you can both read and write the values of both controls and indicators from the block diagram, programmatically. In this case, the control vs. indicator read vs. write distinction is blurred.

PREV NEXT

Block Diagrams

The *block diagram* window holds the graphical source code of LabVIEW VIs. LabVIEW's block diagram corresponds to the lines of text found in a more conventional language like C or BASICit is the actual executable code. You construct the block diagram by wiring together objects that perform specific functions. In this section, we will discuss the various components of a block diagram: *terminals*, *nodes*, and *wires*.

The simple VI shown in Figure 3.2 computes the sum of two numbers. Its diagram in Figure 3.3 shows examples of terminals, nodes, and wires.

Figure 3.2. The front panel of Add.vi contains controls for data input and indicators for data display

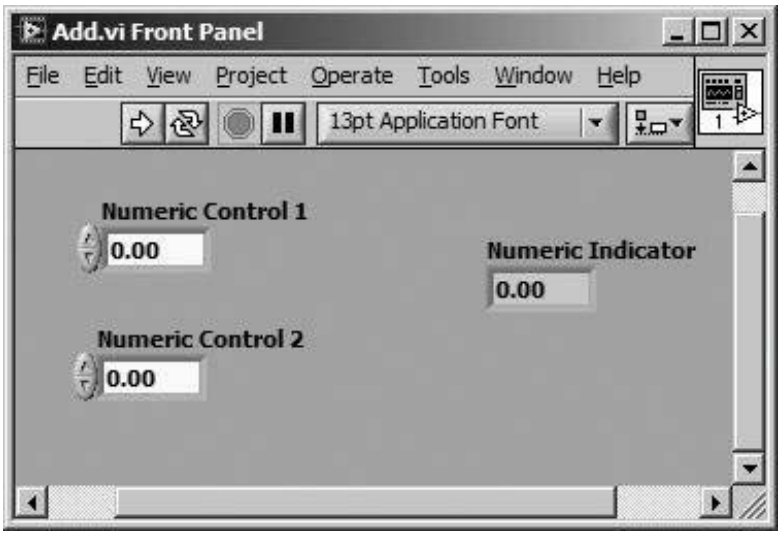
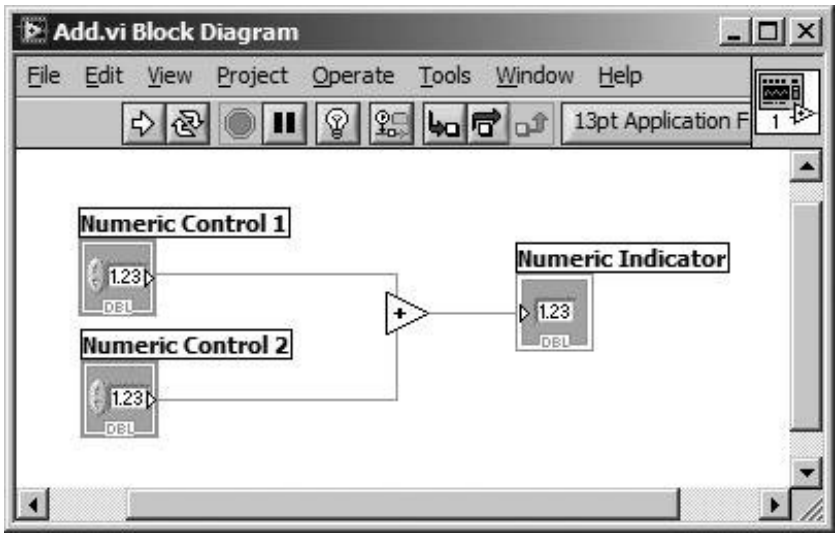


Figure 3.3. The block diagram of Add.vi contains terminals, nodes, and wires (the functional source code)



When you place a control or indicator on the front panel, LabVIEW automatically creates a corresponding *terminal* on the block diagram. By default, you cannot delete a block diagram terminal that belongs to a control or indicator, although you may try to your heart's content. The terminal disappears only when you delete its corresponding control or indicator on the front panel.

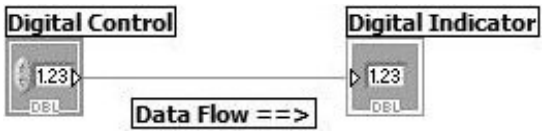
You can allow the deletion of panel terminals on the block diagram by enabling the Delete/copy panel terminals from diagram option in the Block Diagram category of the Tools>>Options dialog (we will learn more about the LabVIEW Options dialog in Chapter 15). However, this is only recommended if you are very comfortable and experienced with editing LabVIEW VIs. It is very easy to mistakenly delete front panel controls and indicators without realizing it, because you may not be paying much attention to the front panel while editing the block diagram.

Control terminals have thick borders along with an arrow pointing out from the right, while indicator borders are thin and have an arrow pointing in from the left. It is very important to distinguish between the two because they are not functionally equivalent (Control=Input=data source and Indicator = Output = data sink, so they are not

interchangeable).

You can think of terminals as entry and exit ports in the block diagram, or as sources and destinations. Data that you enter into `Numeric Control 1` (shown in Figure 3.4) exits the front panel and enters the block diagram through the `Numeric Control 1` terminal on the diagram. The data from `Numeric Control 1` follows the wire and enters the Add function input terminal. Similarly, data from `Numeric Control 2` flows into the other input terminal of the Add function. Once data is available on both input terminals of the Add function, it performs its internal calculations, producing a new data value at its output terminal. The output data flows to the `Numeric Indicator` terminal and reenters the front panel, where it is displayed for the user.

Figure 3.4. Block diagram with a control (input/source) terminal, an indicator (output/sink) terminal, and a wire through which data will flow

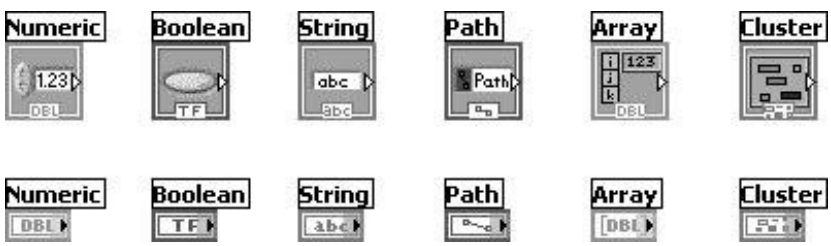


View Terminals as Icons

Block diagram terminals have a View As Icon option (available from their pop-up menus), which causes them to be viewed as *Icons*. A terminal viewed as an icon is larger (than with the setting turned off) and contains an icon that reflects the terminal's front panel control type. With the View As Icon option turned off, a terminal will be more compact and display the data type more predominantly. The functionality is exactly the same for either setting; it's just a matter of preference.

Figure 3.5 shows the terminals of several different front panel controls with the View As Icon option selected (top row) and with the option not selected (bottom row).

Figure 3.5. Terminals with View As Icon setting enabled (top row) and disabled (bottom row)



This setting is configurable for each terminal on the block diagram. You can choose whether terminals will be placed onto the block diagram as icons, by checking the Place front panel terminals as icons setting in the Block Diagram category of the Tools>>Options dialog. This LabVIEW option is turned on, by default.

Nodes

A *node* is just a fancy word for a program execution element. Nodes are analogous to statements, operators, functions, and subroutines in standard programming languages. The Add and Subtract functions represent one type of node. A structure is another type of node. Structures can execute code repeatedly or conditionally, similar to loops and case statements in traditional programming languages. LabVIEW also has special nodes, called Formula Nodes, which are useful for evaluating mathematical formulas or expressions. In addition, LabVIEW has very special nodes called Event Structures that can capture front panel and user-defined events.

Wires

A LabVIEW VI is held together by *wires* connecting nodes and terminals. Wires are the data paths between source and destination terminals; they deliver data from one source terminal to one or more destination terminals.

If you connect more than one source or no source at all to a wire, LabVIEW disagrees with what you're doing, and the wire will appear brokena wire can only have one data source, but it can have multiple data sinks.

This principle of wires connecting source and destination terminals explains why controls and indicators are not interchangeable. Controls are source terminals, while indicators are destinations, or "sinks."

Each wire has a different style or color, depending on the data type that flows through the wire. The block diagram shown in Figure 3.3 depicts the wire style for a numeric scalar valuea thin, solid line. The chart in Figure 3.6 shows a few wires and corresponding types.

Figure 3.6. Basic wire styles used in block diagrams

[View full size image]

To avoid confusing your data types, simply match up the colors and styles!

Dataflow ProgrammingGoing with the Flow

Because LabVIEW is not a text-based language, its code cannot execute "line by line." The principle that governs G program execution is called *dataflow*. Stated simply, a node executes only when data arrives at all its input terminals; the node supplies data to all of its output terminals when it finishes executing; and the data passes immediately from source to destination terminals. Dataflow contrasts strikingly with the control flow method of executing a text-based program, in which instructions are executed in the sequence in which they are written. This difference may take some getting used to. Although traditional execution flow is instruction driven, dataflow execution is data driven or *data dependent*.

PREV NEXT

LabVIEW Projects

LabVIEW Projects allow you to organize your VI and other LabVIEW files as well as non-LabVIEW files, such as documentation and just about anything else you can think of. When you save your project, LabVIEW creates a project file (`.lvproj`). In addition to storing information about the files contained in your project, the project file stores your project's configuration, build, and deployment information.

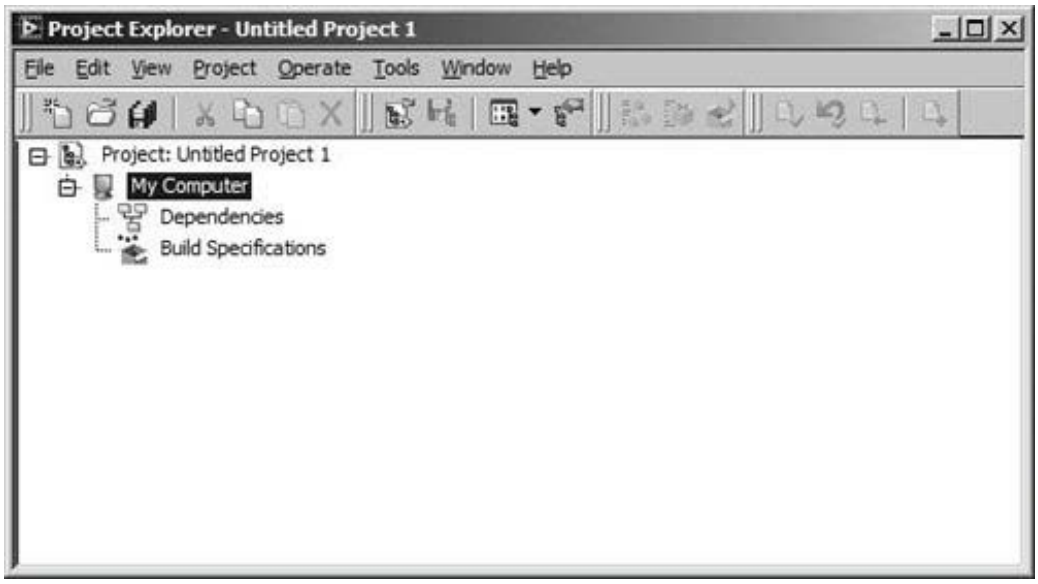
You might be asking yourself: "Why do I need a project?" There are a lot of reasons, but a better question might be to ask: "When don't I need a project?" The answer to this is very simple. You don't need a project if you are only going to create one or two VIs and you are more interested in collecting and analyzing data, than you are in the VIs that you use to collect and analyze data. However, if you are interested in managing your VIs as software, you should organize your VIs in a LabVIEW Project.

Project Explorer Window

The Project Explorer window is where you will create and edit your LabVIEW Projects. Figure 3.7 shows an empty project. You can create an empty project by selecting File>>New . . . from the menu and then selecting Empty Project from the dialog box.

Figure 3.7. Project Explorer window showing a new, empty LabVIEW project

[View full size image]



The project appears as a tree with several items. The root item is the Project root, which shows the name of the project file and contains all of the project items. The next item is My Computer, which represents the local computer as a target in the project.

A target is a location where your VIs will be deployed. A target can be your local computer, a LabVIEW RT controller, a Personal Digital Assistant (PDA), a LabVIEW FPGA device, or anywhere you can run LabVIEW VIs. You can add targets to your project by right-clicking on the project root and selecting New>>Targets and Devices from the shortcut menu. In order to be able to add additional targets to your project, you will need to install the appropriate LabVIEW add-on module. For example, the LabVIEW Real-Time, FGPA, and PDA modules allow you to add those targets to your project.

Beneath the My Computer target are Dependencies and Build Specifications. Dependencies are items that VIs in the project require. Build specifications are the rules that define how your application will be deployed.

Project Explorer Toolbars

The Project Explorer has several toolbars that allow you to easily perform common operations. These are the Standard, Project, Build, and Source Control toolbars, shown in Figure 3.8 (respectively, from left to right).

Figure 3.8. Project Explorer toolbars

[View full size image]

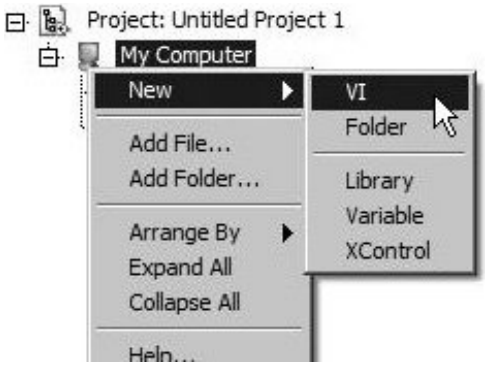
You can choose which of these toolbars are visible from the View>>Toolbars menu, or by right-clicking on the toolbar and selecting the desired toolbar from the pop-up menu that appears (see Figure 3.9).

Figure 3.9. Toolbar pop-up menu showing which toolbars are visible (checked)

Adding Items to Your Project

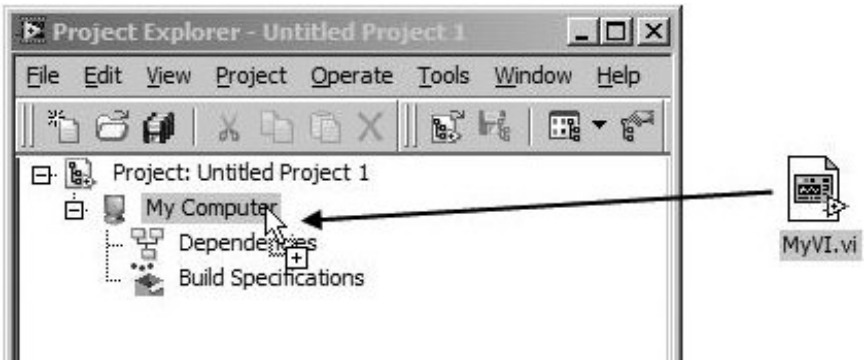
You can add items to your project, beneath the My Computer target. You can also create folders, to better organize the items in a project. There are a variety of ways to add items to your project. The pop-up shortcut menu is probably the easiest way to add new VIs and create new folders, as shown in Figure 3.10.

Figure 3.10. Adding a new VI to a LabVIEW project from the pop-up menu of My Computer in the Project Explorer



You can also add items from the pop-up menu, but the easiest way (in Windows and Mac OS) is to drag the item or directory from disk into the Project Explorer, as shown in Figure 3.11. You can also drag a VI's icon (in the upper-right corner of a front panel or block diagram window) to the target.

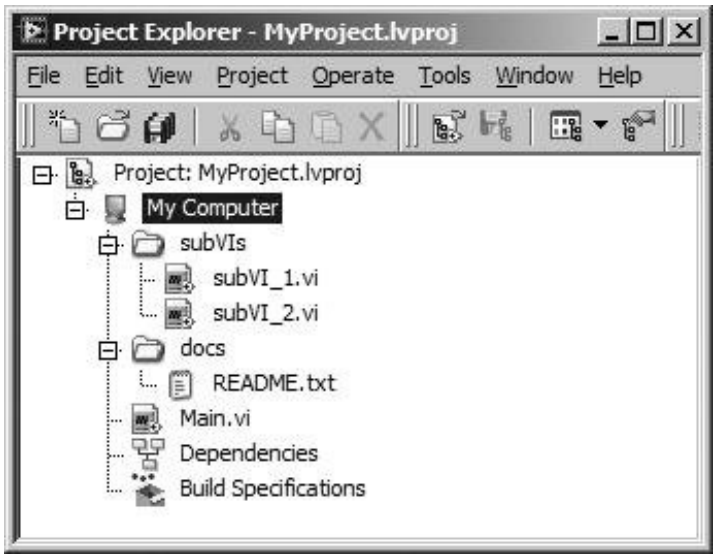
Figure 3.11. Drag and drop to add a VI into project



Project Folders

Project folders are used to organize your project files. For example, you could create a folder for your subVIs and another folder for your project documentation, as shown in Figure 3.12.

Figure 3.12. The Project Explorer showing project folders used to organize project files

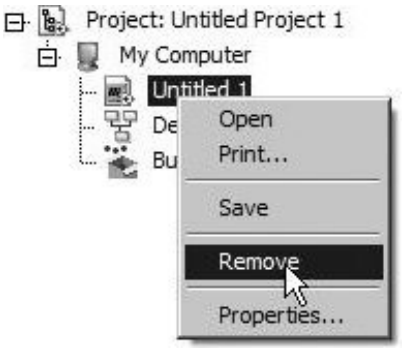


Project folders are virtual, which means that they do not necessarily represent folders on disk (although you might choose to create project folders that reflect the organization of your folders on disk). One side effect of this is that after you add a directory on disk to a project, LabVIEW will not automatically update the project folder contents to reflect changes made to the directory on disk. You will have to synchronize these manually, if you wish to do so.

Removing Items from a Project

Items can be removed from the project by right-clicking the item in the Project Explorer and selecting Remove from the pop-up shortcut menu, as shown in Figure 3.13.

Figure 3.13. Removing items from a project using the pop-up menu



Alternatively you can delete an item by selecting the item in the Project Explorer and then pressing the <Delete> key or pressing the Delete button on the Standard toolbar.

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

Building Applications, Installers, DLLs, Source Distributions, and Zip Files

The project environment provides you with the ability to create built software products from your VIs. To do this, pop up on the Build Specifications node in the Project Explorer window and select a build output type from the New>> submenu. You can choose from the following options:

- Application Use stand-alone applications to provide other users with executable versions of VIs. Applications are useful when you want users to run VIs without installing the LabVIEW development system. Windows applications have an `.exe` extension, Mac OS X applications have an `.app` extension, and Linux applications have no file extension.
- Installer (Windows only) Use installers to distribute stand-alone applications, shared libraries, and source distributions that you create with the Application Builder. Installers that include the LabVIEW Run-Time Engine are useful if you want users to be able to run applications or use shared libraries without installing LabVIEW.
- Shared Library Use shared libraries if you want to call VIs using text-based programming languages, such as NI LabWindows/CVI, Microsoft Visual C++, and Microsoft Visual Basic. Using shared libraries provides a way for programming languages other than LabVIEW to access code developed with LabVIEW. Shared libraries are useful when you want to share the functionality of the VIs you build with other developers. Other developers can use the shared libraries but

cannot edit or view the block diagrams unless you enable debugging. Windows shared libraries have a `.dll` extension. Mac OS X shared libraries have a `.framework` extension. Linux shared libraries have an `.so` extension.

- Source Distribution Use source distributions to package a collection of source files. Source distributions are useful if you want to send code to other developers to use in LabVIEW. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings. You also can select different destination directories for VIs in a source distribution without breaking the links between VIs and subVIs.
- Zip File Use zip files when you want to distribute files or an entire LabVIEW project as a single, portable file. A zip file contains compressed files, which you can send to users. Zip files are useful if you want to distribute instrument driver files or selected source files to other LabVIEW users. You also can use the Zip VIs (found on the Programming>>File I/O>>Zip palette) to create zip files programmatically.

Each of these build output types has its own set of build specifications. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for directories of VIs.

Building an application and building an installer are two separate steps. You will need to create one build specification for each of these outputs.

If you select Source Distribution from the File>>Save As . . . dialog, you will be guided through the process of creating a Source Distribution. This requires the creation of a new project file, if the source VI is not part of a LabVIEW project.

For more information on building applications and using build specifications in LabVIEW projects, refer to the Fundamentals>>Organizing and Managing a Project>>Concepts>>Using Build Specifications section of the LabVIEW Help.

More Project Features

You have just learned about the fundamental features of the LabVIEW project environment. This knowledge will help you organize your VIs and project files and allow you to work efficiently. However, there are many more features of the LabVIEW project environment. For more information on using the LabVIEW Project Explorer, refer to the Fundamentals>>Organizing and Managing a Project section of the LabVIEW Help documentation.

PREV NEXT

SubVIs, the Icon, and the Connector

A subVI simply refers to a VI that is going to be called by another VI. Any VI can be configured to function as a subVI. For example, let's say you create a VI called Mean.vi that calculates the mean value of an array. You can always run Mean.vi from its front panel (by pressing the run button on the toolbar), but you can also configure Mean.vi so that other VIs can call it as a function on their block diagram (they call Mean.vi as a subVI).

When your VI operates as a subVI, its controls and indicators receive data from and return data to the VI that calls it. A VI's *icon* represents it as a subVI in the block diagram of another VI. An icon can include a pictorial representation or a small textual description of the VI, or a combination of both.

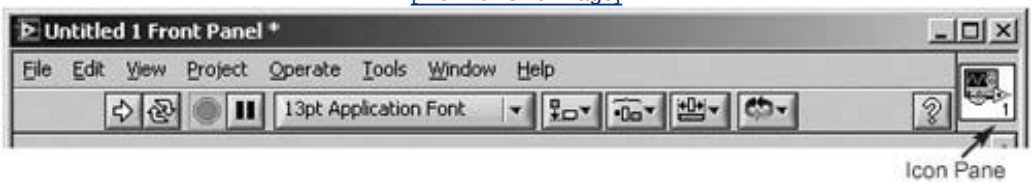
The VI's *connector* functions much like the parameter list of a C or Pascal function call; the connector terminals act like little graphical parameters to pass data to and from the subVI. Each terminal corresponds to its very own control or indicator on the front panel. During the subVI call, the input parameter terminals are copied to the connected controls, and the subVI executes. At completion, the indicator values are copied to the output parameter terminals.

Figure 3.14. An icon and its underlying connector

Every VI has a default icon, which is displayed in the icon pane in the upper-right corner of the panel and diagram windows. The default icon is depicted in Figure 3.15.

Figure 3.15. The VI icon pane in the upper-right corner of a VI front panel

[View full size image]



A VI's connector is hidden under the icon; access it by choosing Show Connector from the front panel icon pane pop-up menu (we'll talk more about pop-up menus later). When you show the connector for the first time, LabVIEW helpfully suggests a connector pattern that has twelve terminals (six on the left for inputs and six on the right for outputs). The default connector pane is depicted in Figure 3.16. You can select a different pattern if you desire, and you can assign up to 28 terminals before you run out of real estate on the connector.

Figure 3.16. The VI connector pane in the upper-right corner of a VI front panel

[View full size image]

PREV NEXT

Activity 3-1: Getting Started

Okay, you've read enough for now. It's time to get some hands-on experience. Go ahead and launch LabVIEW. You'll step through the creation of a simple LabVIEW VI that generates a random number and plots its value on a waveform chart. You'll learn more in the next chapter about the steps you'll be taking; for now, just get a feel for the environment.

If you're using the full version of LabVIEW, just launch it and you'll be ready to start building your first VI.

If you're using the evaluation version of LabVIEW, you can still do these activities, because the evaluation version of LabVIEW has no restrictions on creating and editing VIs. Just be aware that the evaluation version of LabVIEW stops working after 30 days. After that point, you'll need to buy a license key to activate your LabVIEW software.

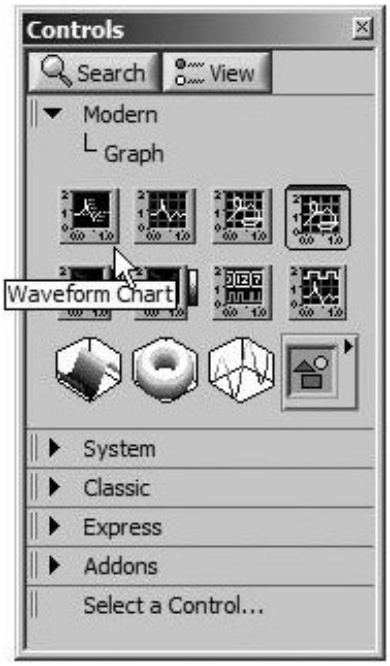
If you're not comfortable working through the activities in this chapter without more background information or have trouble getting them to work, read Chapter 4, "LabVIEW Foundations," and then come back and try again.

1. Launch LabVIEW (if it's been open, you can quit it first).

2. At the LabVIEW *Getting Started* dialog, click Blank VI to create a new blank VI (the Blank VI option is found under the New heading, and you have to click on the "Blank VI" text, not the icon). An "Untitled 1" VI front panel will appear on your screen.

   Go to the floating Controls palette and navigate (by clicking) to the Modern>>Graph subpalette, shown in Figure 3.17. If the Controls palette isn't visible, select View>>Controls Palette from the menu to make it visible. Also, make sure the front panel window is active, or you will see the Functions palette instead of the Controls palette.

Figure 3.17. Modern>>Graph palette showing the Waveform Chart and various other charts and graphs



On the Graph subpalette, select Waveform Chart by clicking it with the mouse button. You will notice that, as you run the cursor over the icons in the Controls palette and subpalettes, the selected button or icon's name appears in a tip strip, as shown in Figure 3.17.

Positioning Tool

You will see the outline of a chart with the cursor "holding" it, as shown in Figure 3.18. Position the cursor in a desirable spot on your front panel and click. The chart magically appears exactly where you placed it, as shown in Figure 3.19. If you want to move it, select the Positioning tool from the Tools palette, and then drag the chart to its new home. If the Tools palette isn't visible, select View>>Tools Palette from the menu.

Figure 3.18. Waveform Chart after it has been dragged onto the front panel (and before it has been dropped)

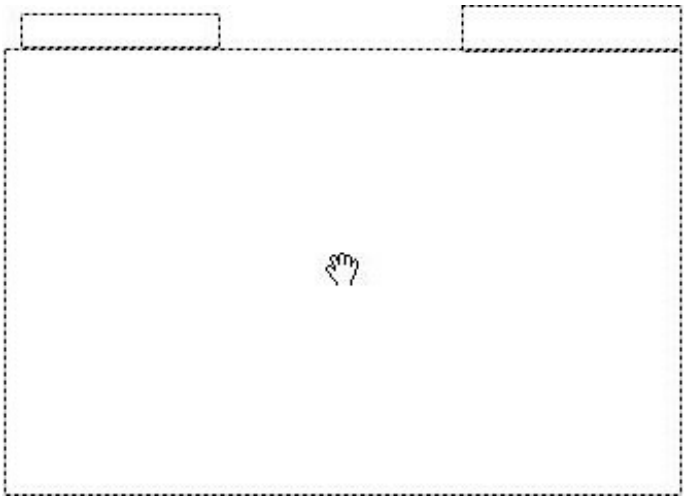
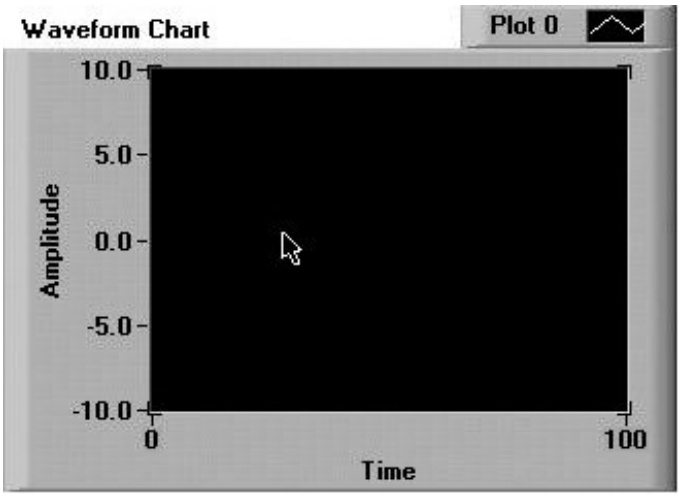
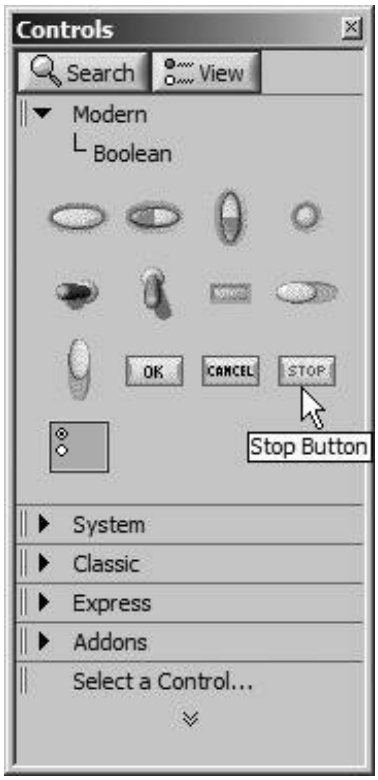


Figure 3.19. Waveform Chart after it has been dropped onto the front panel



3. Go back to the Modern subpalette; then navigate into Boolean subpalette, and choose Stop Button (see Figure 3.20).

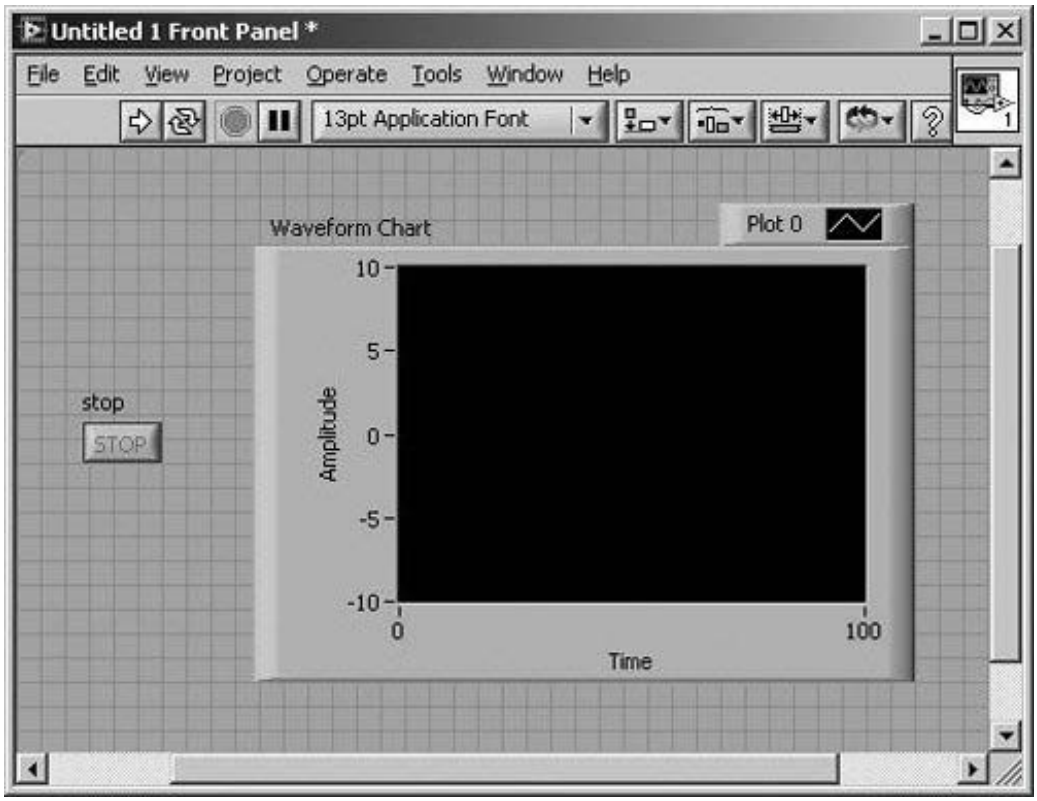
Figure 3.20. Modern>>Boolean palette showing the Stop Button and other Boolean controls and indicators



Place it next to the chart, as shown in Figure 3.21.

Figure 3.21. Your VI front panel, after adding a Waveform Chart and Stop Button

[View full size image]



4. Now change the chart's *Y*-axis scale range from -10 thru +10 to 0 thru 1. Highlight the number "10" (*Y*-axis range max) by click-dragging or by double-clicking on it with the Text Edit tool. Now type in `1.0` and click on the enter button that appears in the Toolbar at the top of the window. Change -10 (*Y*-axis range min) to 0 in a similar manner.

Enter Button

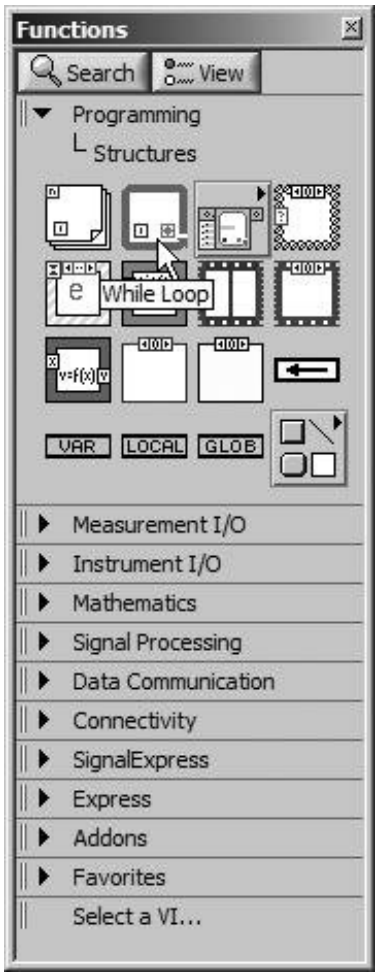
5. Switch to the block diagram by selecting Show Block Diagram from the Window menu. You should see two terminals already there, as shown in Figure 3.22.

Figure 3.22. Your VI block diagram, showing the Stop Button and Waveform Chart terminals

6. Now you will put the terminals inside a While Loop to repeat execution of a segment of your program. Go to the Programming>>Structures subpalette of the floating Functions palette and select the While Loop (see Figure 3.23). Make sure the block diagram window is active, or

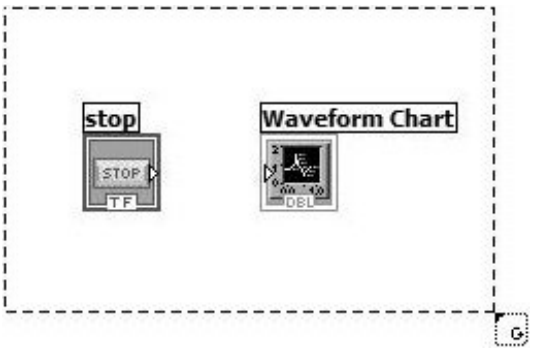
you will see the Controls palette instead of the Functions palette.

Figure 3.23. Programming>>Structures palette showing a While Loop and other LabVIEW structures



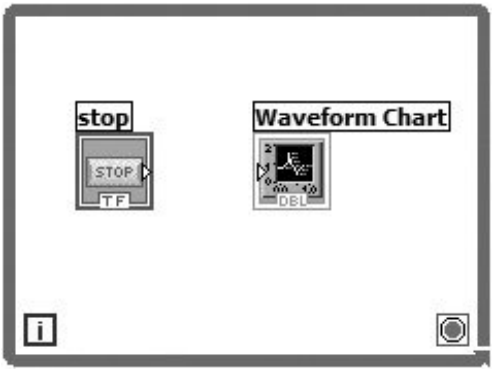
Your cursor will change to a little loop icon. Now enclose the DBL and TF terminals: Click and hold down the mouse button while you drag the cursor from the upper-left to the lower-right corners of the objects you wish to enclose (see Figure 3.24).

Figure 3.24. Dragging the mouse cursor to place a While Loop on the block diagram around a region of code



When you release the mouse button, the dashed line that is drawn as you drag will change into the While Loop border (see Figure 3.25). Make sure to leave some extra room inside the loop.

Figure 3.25. A While Loop containing the code that was enclosed by the dragged region



7. Go to the Functions palette and select Random Number (0-1) from the Programming>>Numeric subpalette. Place it inside the While Loop.

   The While Loop is a special LabVIEW structure that repeats the code inside its borders until the *conditional terminal* reads a TRUE value (when configured to *Stop if True*, appearing as a small red stop sign). It is the equivalent of a Do-While Loop in a more conventional language.

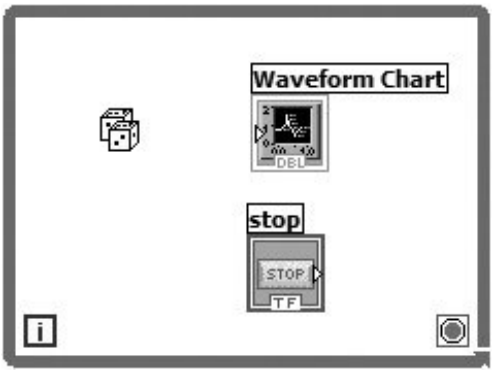
You can configure the conditional terminal of the While Loop to Continue if True (appearing as a small green loop), from its pop-up menu or by clicking on it with the

Operating tool. You'll learn more about While Loops in Chapter 6, "Controlling Program Execution with Structures."

8. With the Positioning tool active, arrange your diagram objects so that they look like the block diagram in Figure 3.26.

Positioning Tool

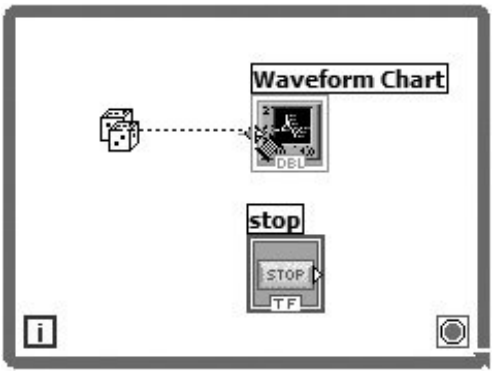
Figure 3.26. Your block diagram after placing the Random Number (0-1) function



9. With the Wiring tool active, click once on the Random Number (0-1) icon, drag the mouse over to the `Waveform Chart`'s terminal, and click again (see Figure 3.27).

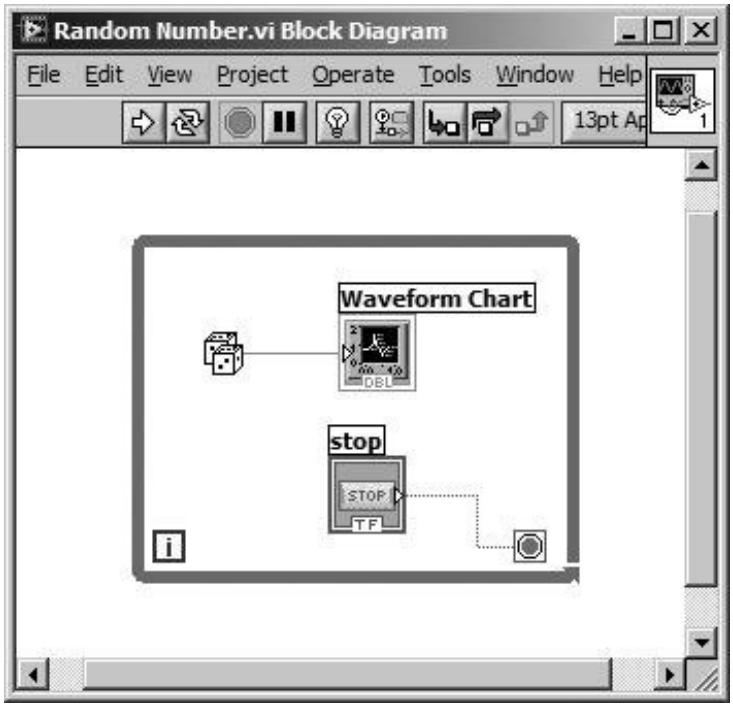
Wiring Tool

Figure 3.27. Your block diagram, as you drag a wire from the Random Number (0-1) function to the Waveform Chart terminal



You should now have a solid orange wire connecting the two icons, as shown in Figure 3.28. If you mess up, you can select the wire or wire fragment with the Positioning tool and then hit the <delete> key to get rid of it. Now wire the Boolean TF terminal to the conditional terminal of the While Loop. The loop will execute while the switch on the front panel is FALSE (not pressed down) and stop when the switch becomes TRUE (pressed down).

Figure 3.28. Your block diagram after completing the wiring tasks



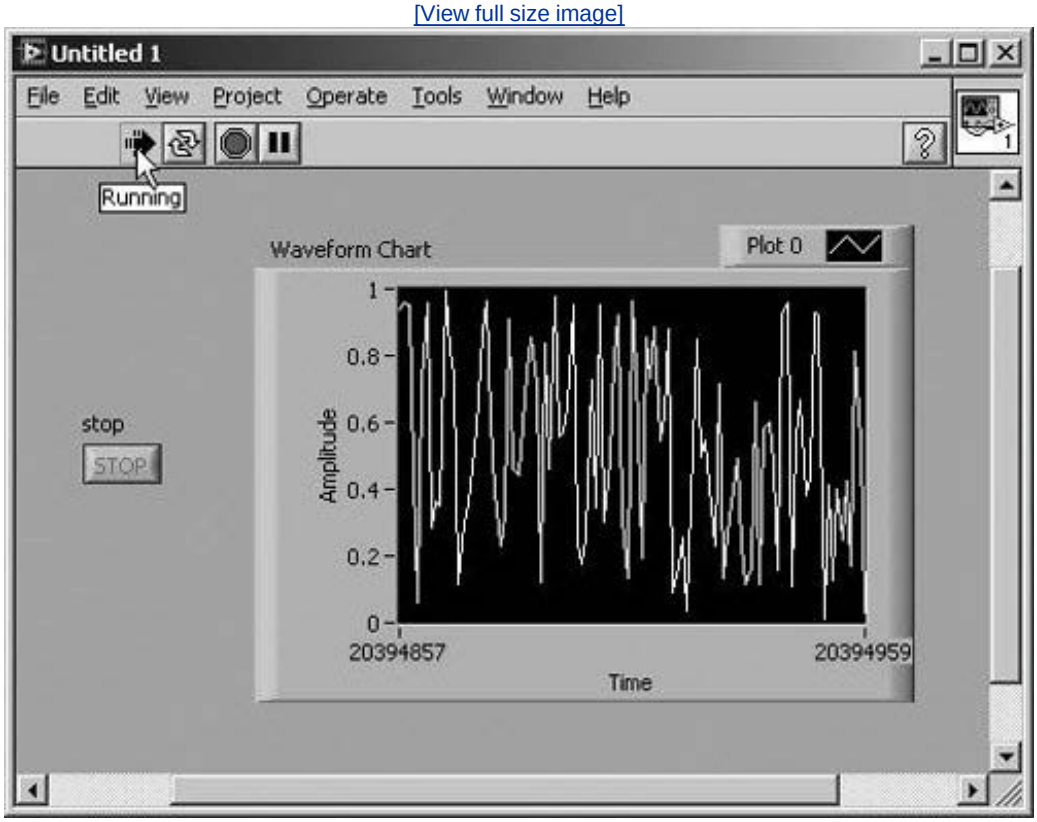
10. You should be about ready to run your VI. First, switch back to the front panel by selecting Show Front Panel from the Window menu. Now click on the run button to run your VI. You will see a series of random numbers plotted continuously across the chart. When you want to stop, click on the stop Boolean button using the Operating tool (see Figure 3.29).

Run Button

Operating Tool

Figure 3.29. Your VI front panel after pressing the Run button

[View full size image]



11. Create a directory or folder called `MYWORK` in a convenient location (such as your home or documents directory). Save your VI in your `MYWORK` directory or folder by selecting Save from the File menu and pointing out the proper location to save to. Name it Random Number.vi.

Save all of your subsequent activities in `MYWORK` *so you can find them easily!*

Remember if you get stuck or just want to compare your work, the solutions to every activity in this book can be found in the `EVERYONE` *directory or folder on the accompanying CD. You can view them in the sample version or the full version of LabVIEW.*

Congratulate yourselfyou've just written your first LabVIEW program! Don't worry that it doesn't actually DO much; your programs will be more powerful and have more of a purpose soon enough!

PREV NEXT

Alignment Grid

In the example you just completed, you probably noticed the grid lines on the VI's front panel and how the waveform chart and stop controls "snap" to the grid lines as you move them around with the mouse. This feature is very useful for keeping your front panel objects aligned. If you do not like this feature, you can turn it off in the LabVIEW Options dialog. (Figures 3.30 and 3.31 show a VI front panel with the alignment grid feature turned ON and OFF, respectively.) Open the options dialog by selecting Tools>>Options. . . from the menu. Navigate to the Alignment Grid category. From here, you can choose to show or hide the grid lines and to enable or disable grid alignment. If you are not sure of your preference, leave the alignment grid onyou'll probably like it.

Figure 3.30. VI with alignment grid option ON

[View full size image]

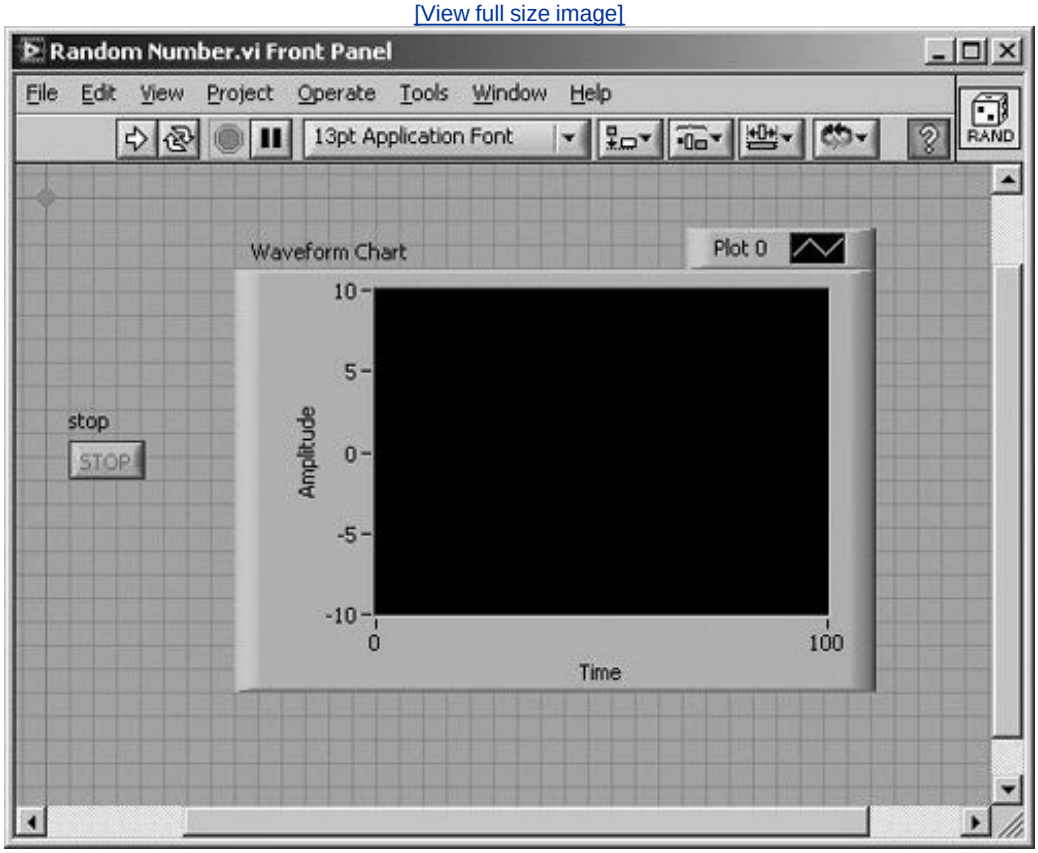
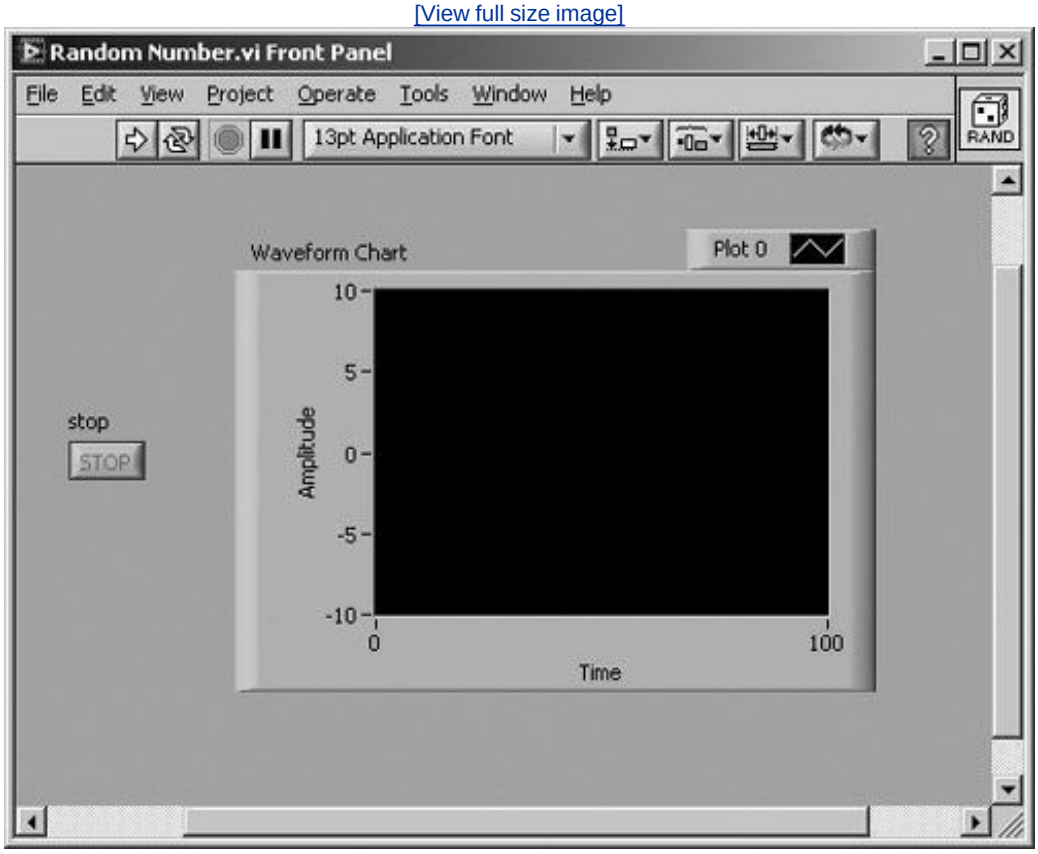


Figure 3.31. VI with alignment grid option OFF

[View full size image]



Throughout the remainder of the book, all the figures show front panels with the alignment grid turned off. This helps to make sure that you can easily distinguish the features of the objects on the front panel.

PREV NEXT

Pull-Down Menus

Keep in mind that LabVIEW's capabilities are many and varied. This book by no means provides an exhaustive list of all of LabVIEW's ins and outs (it would be several thousand pages long if that were the case); instead, we try to get you up to speed comfortably and give you an overview of what you can do. If you want to know everything there is to know about a subject, we'd recommend looking it up in one of LabVIEW's many manuals, attending a seminar, or going to ni.com/labview on the Web. See Appendix E, "Resources for LabVIEW," for an exhaustive list of other resources. Feel free to skim through this section and some of the subsequent ones, but remember that they're here if you need a reference.

LabVIEW has two main types of menus: pull-down and pop-up. You used some of them in the last activity, and you will use both extensively in all of your program development henceforth. Now you will learn more about what they can do. We'll cover pull-down menu items very briefly in this section. You might find it helpful to look through the menus on your computer as we explain them, and maybe experiment a little.

The menu bar at the top of a VI window contains several pull-down menus (in Mac OS X, the menu bar will be at the top of the screen, consistent with other Mac OS X applications). When you click on a menu bar item, a menu appears below the bar. The pull-down menus contain items common to many applications, such as Open, Save, Copy, and Paste, and many other functions particular to LabVIEW. We'll discuss some basic pull-down menu functions here. You'll learn more about the advanced capabilities later.

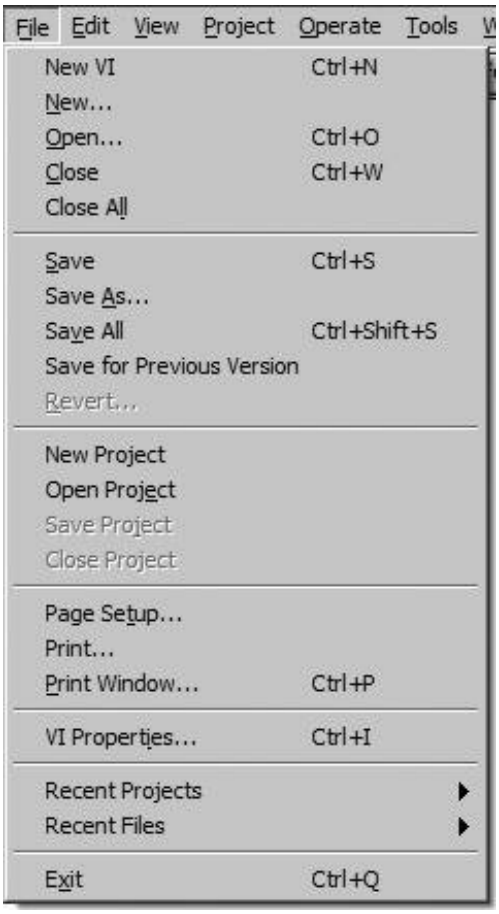
Many menus also list shortcut keyboard combinations for you to use if you choose. To use keyboard shortcuts, press the appropriate key in conjunction with the <control> key on PCs, the <command> key on Macs, and the <meta> key on Linux.

Many of the menu items show keyboard shortcuts to the right of their corresponding commands. You may want to use the shortcuts instead of the menus. You can customize the menu shortcuts in the Tools>>Options dialog under the Menu Shortcuts section.

File Menu

Pull down the File menu (see Figure 3.32), which contains commands common to many applications, such as Save and Print. You can also create new VIs or open existing ones from the File menu. In addition, you can show VI Properties information and development history from this menu.

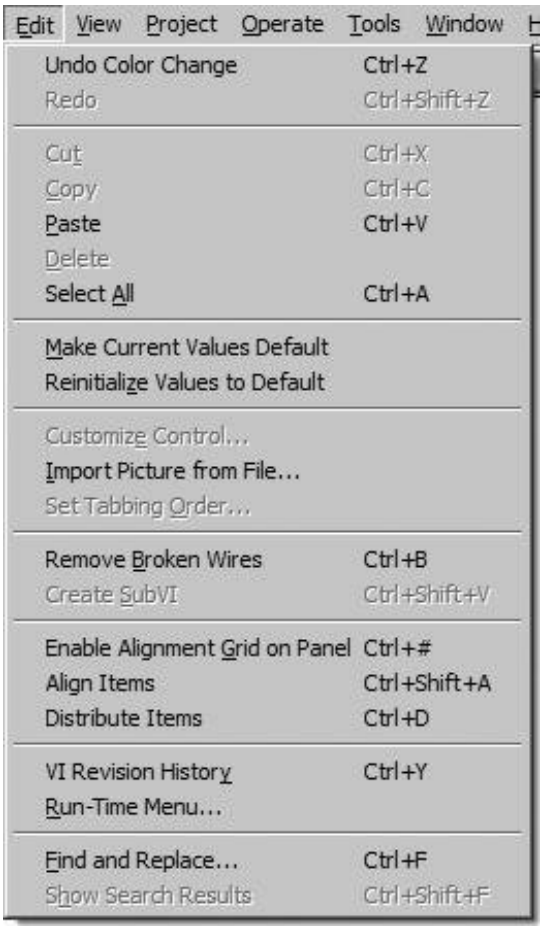
Figure 3.32. File menu



Edit Menu

Take a look at the Edit menu (see Figure 3.33). It has some universal commands, like Undo, Cut, Copy, and Paste, that let you edit your window. You can also search for objects with the Find and Replace . . . command and remove bad wires from the block diagram.

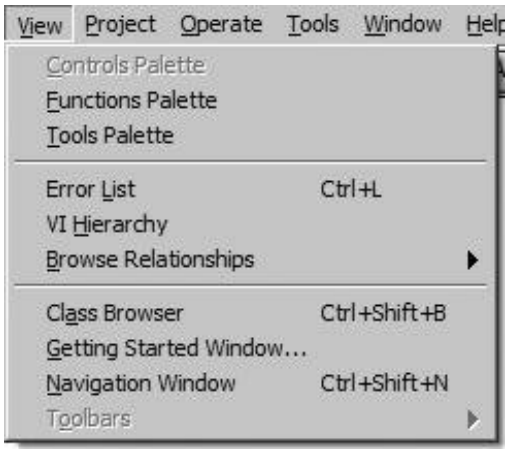
Figure 3.33. Edit menu



View Menu

In the View menu (see Figure 3.34), you will see options for opening the Controls Palette, Functions Palette, and Tools Palette if you've closed them. You can also show the error list and see a VI's hierarchy. The Browse Relationships submenu contains features to simplify navigation among large sets of VIs, such as determining all of a VI's subVIs and where a VI is used as a subVI.

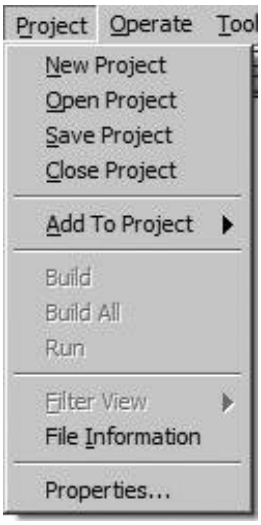
Figure 3.34. View menu



Project Menu

The Project menu (see Figure 3.35) allows you to open a LabVIEW project or create a new project, as well as operate on the project to which the active VI belongs. If the active VI does not belong to any LabVIEW project, only the Open Project and New Project menu items will be enabled.

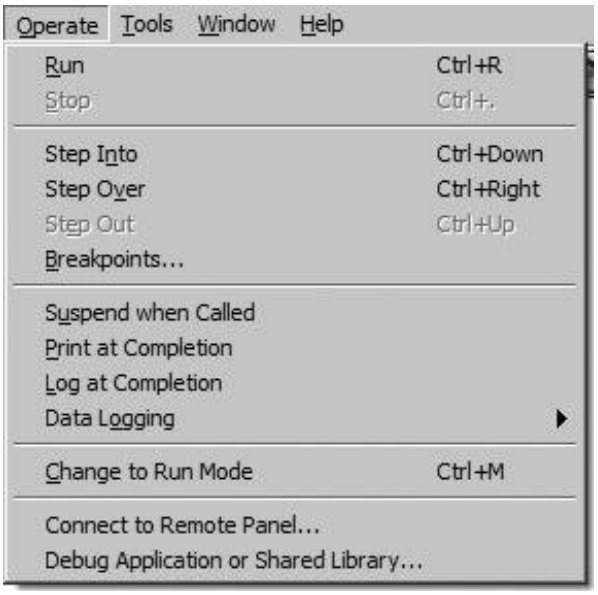
Figure 3.35. Project menu



Operate Menu

You can run or stop your program from the Operate menu (see Figure 3.36), although you'll usually use Toolbar buttons. You can also change a VI's default values, control "print and log at completion" features, and switch between run mode and edit mode.

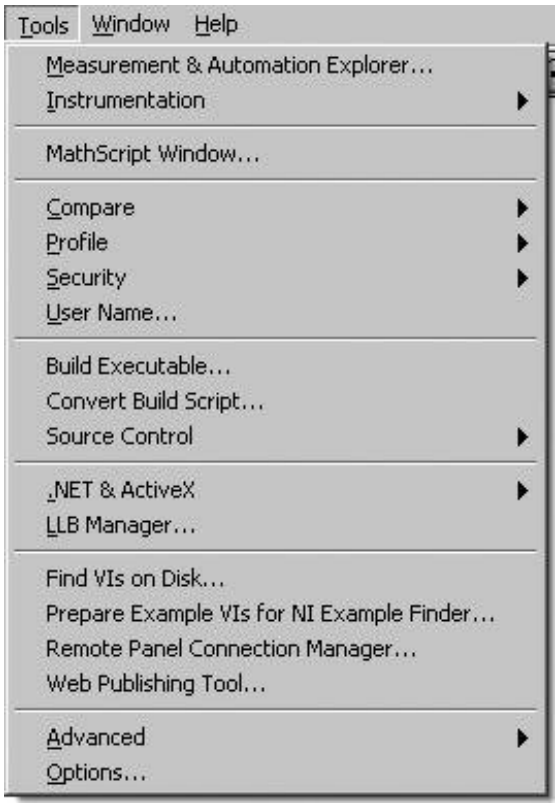
Figure 3.36. Operate menu



Tools Menu

The Tools menu (see Figure 3.37) lets you access built-in and add-on tools and utilities that work with LabVIEW, such as the Measurement & Automation Explorer, where you configure your DAQ devices, or the Web Publishing Tool for creating HTML pages from LabVIEW. You can view and change the myriad of LabVIEW's Options . . ..

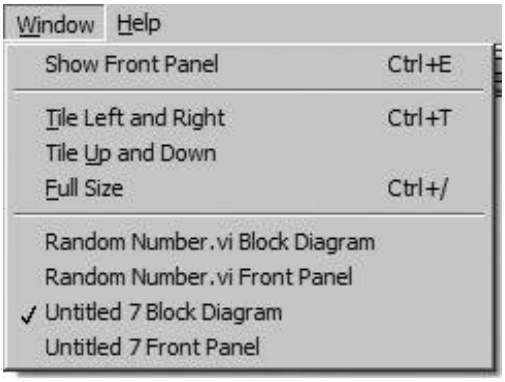
Figure 3.37. Tools menu



Window Menu

Pull down the Window menu (see Figure 3.38). Here you can toggle between the front panel and block diagram windows, "tile" both windows so you can see them at the same time, and switch between open VIs.

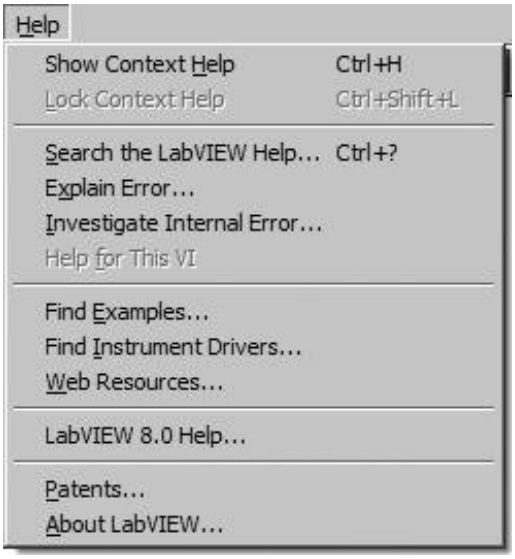
Figure 3.38. Window menu



Help Menu

You can show, hide, or lock the contents of the Context Help window using the Help menu (see Figure 3.39). You can also access LabVIEW's online reference information and view the About LabVIEW information window.

Figure 3.39. Help menu



PREV NEXT

Floating Palettes

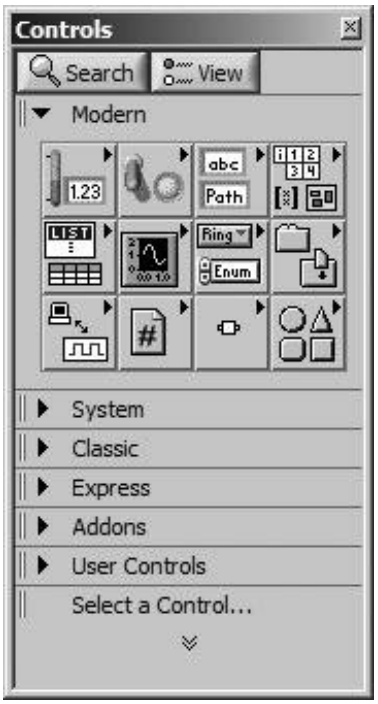
LabVIEW has three often-used floating palettes that you can place in a convenient spot on your screen: the Tools palette, the Controls palette, and the Functions palette. You can move them around by clicking on their title bar and dragging. Close them just like you would close any window in your operating system. If you decide you want them back, select the palette you want from the View menufor example, View>>Tools Palette opens the Tools palette.

Controls and Functions Palettes

You will be using the Controls palette a lot, because that's where you select the controls and indicators that you want on your front panel. You will probably use the Functions palette even more often, because it contains the functions and structures used to build a VI.

The Controls and Functions palettes are unique in several ways. Most importantly, the Controls palette is only visible when the front panel window is active, and the Functions palette is only visible when the block diagram window is active. Both palettes have subpalettes containing the objects you need to access. As you pass the cursor over each subpalette button in the Controls and Functions palettes, you will notice that the subpalette's name appears in a tip strip beneath the mouse pointer (see Figure 3.40).

Figure 3.40. Controls palette



To select an object in the subpalette, click the mouse button over the object, and then click on the front panel or block diagram to place it where you want it.

Palette Item Categories

At the top level of the palettes are several "Categories," such as *Modern*, *System*, *Classic*, *Express*, *Addons*, *Favorites*, and others. You can expand and close these categories, by clicking on them, to navigate the tree of items and subcategories that are available within the palette.

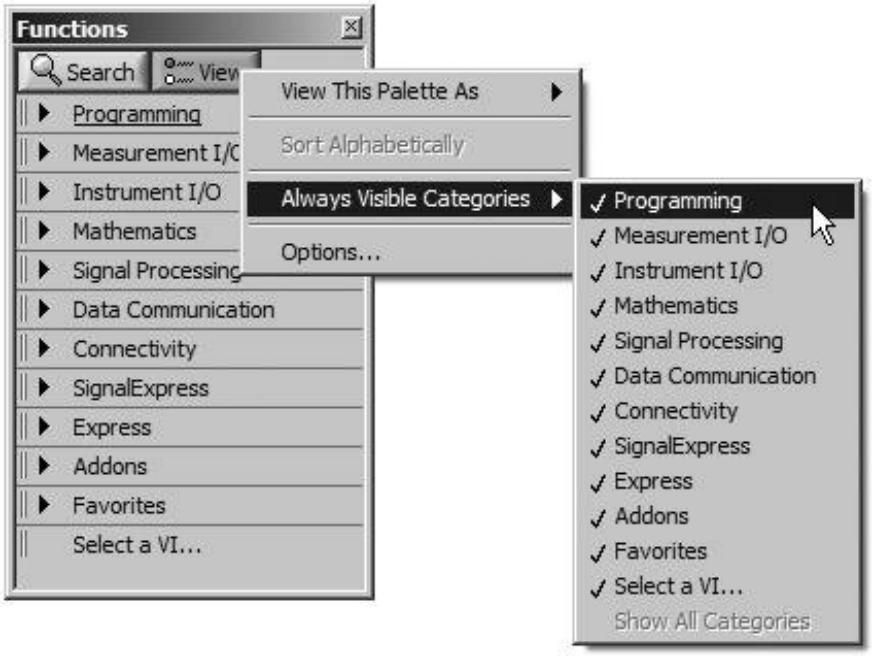
One great category is Favorites (this is only present on the Functions palette, not the Controls palette). You can use this to group together items that you access frequently. When you find a function, structure, VI, or subpalette that you really like, right-click on the object and select Add Item to Favorites from the shortcut menu. This will add the object to the Favorites category, for quick and easy access.

Showing and Hiding Palette Categories

You can choose which categories you want to appear on the Controls or Functions palettes by pressing the View button and then selecting categories from the Always Visible Categories submenu. If you want all of the categories to be visible, select the Show All Categories option from the Always Visible Categories submenu (see Figure 3.41).

Figure 3.41. The Always Visible Categories submenu of the palette View

button



If a category is not selected in the Always Visible Categories, then you will not normally see it in the palette. However, you can temporarily display all of the categories by clicking on the double "down" arrows at the very bottom of the palette, as shown in Figure 3.42. You can hide those categories again by clicking on the double "up" arrows at the very bottom of the palette, as shown in Figure 3.43.

Figure 3.42. Functions palette with only the "always visible" categories visible

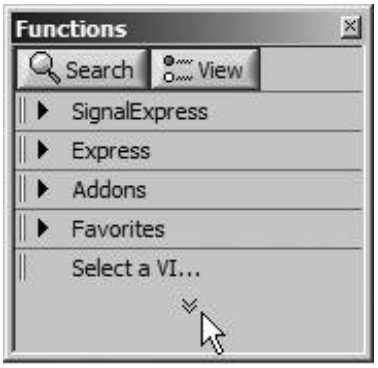
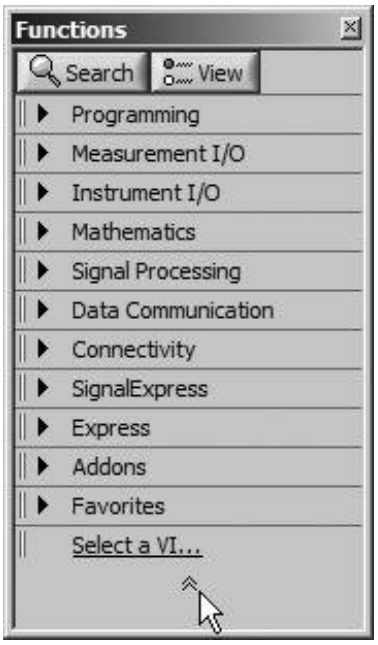


Figure 3.43. Functions palette with all categories visible



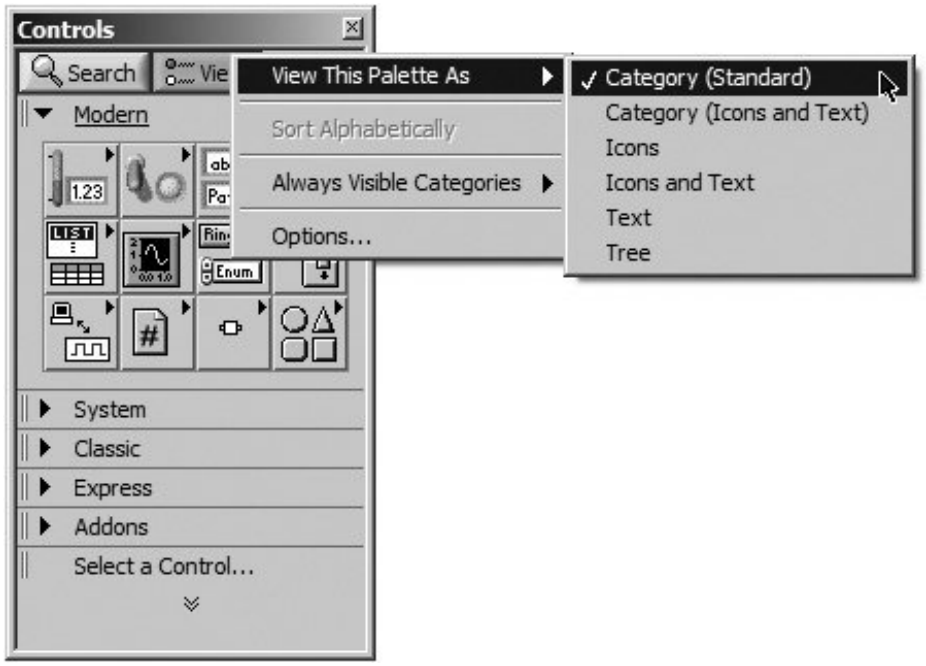
Reordering Palette Categories

You can reorder the palette categories by right-clicking on them and choosing Move this Category Up, Move this Category Down, or Move to Top (Expand by Default). Additionally, you can drag and drop the categories within the list by clicking on the two vertical grab-bars (||) to the left of the category text.

Palette View Formats

The way that you navigate the palettes can be changed by choosing a different palette View Format. Pressing the View button at the top of a palette will open a menu. In the View This Palette As submenu, you can choose from six different *View Formats*, as shown in Figure 3.44.

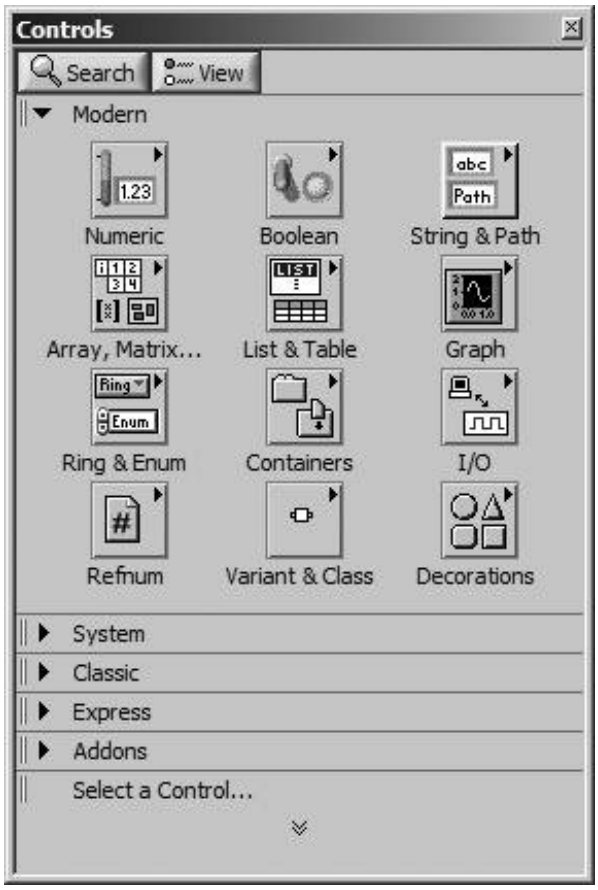
Figure 3.44. The View This Palette As submenu of the palette View button menu showing the currently selected (checked) palette view format



The following View Formats are available:

- Category (Standard) is the default View Format, and the one shown throughout this book.
- Category (Icons and Text), shown in Figure 3.45, is similar to Category (Standard), except that each item's name appears directly beneath its icon.

Figure 3.45. "Category (Icons and Text)" palette view format



- Icons, shown in Figure 3.46, is the format that most are familiar with from previous versions of LabVIEW. Each subpalette and item is represented by an icon. When you mouse-over an item, its name appears at the top of the palette.

Figure 3.46. "Icons" palette view format

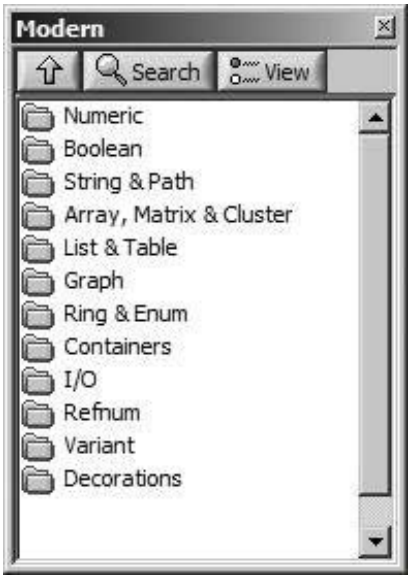
- Icons and Text, shown in Figure 3.47, is similar to Icons, except that each item's name

appears directly beneath its icon.

Figure 3.47. "Icons and Text" palette view format

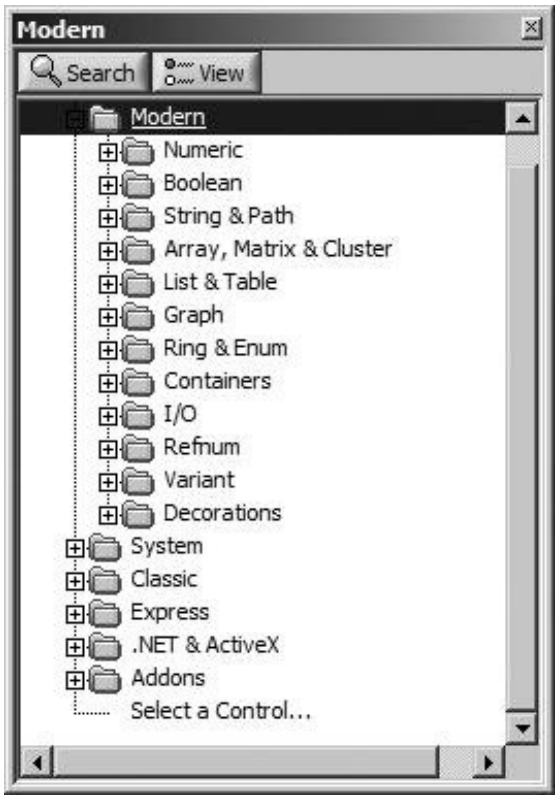
- Text, shown in Figure 3.48, is minimal. It behaves like the Icons and Icons and Text formats, where clicking on a subpalette navigates down into that subpalette. Subpalettes are represented by folders with names, and items are represented by their name.

Figure 3.48. "Text" palette view format



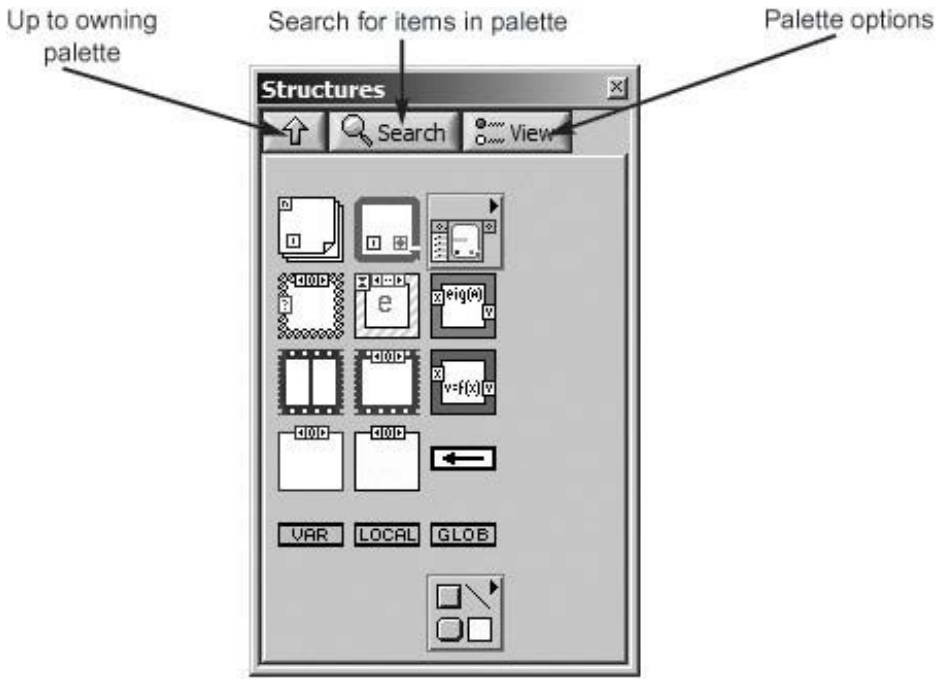
- Tree, shown in Figure 3.49, is also minimal, having only folder icons and item names. However, it behaves more like the Category formats, having a tree hierarchy.

Figure 3.49. "Tree" palette view format



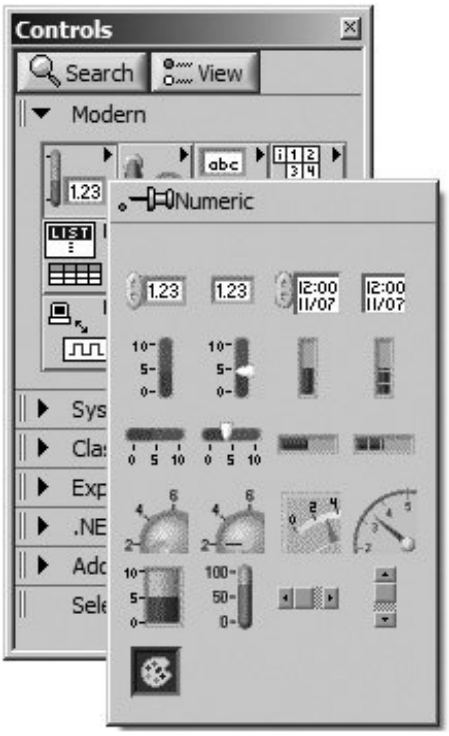
The "Icons," "Icons and Text," and "Text" palette view formats all have an "up" button (see Figure 3.50), which, when pressed, returns you to the previous ("owning") palettesince these view formats are not a tree view. You can search for a specific item in a palette by clicking on the spyglass icon (see Figure 3.50).

Figure 3.50. The buttons at the top of each palette are used for navigation and configuration options.



There is another way to navigate palettes that some people find a little easier. Instead of each subpalette replacing the current palette, you can pass through subpalettes in a hierarchical manner without them replacing their parent palettes. You can do this by right-clicking (Windows) or control-clicking (Mac OS X) the subpalette icons (see Figure 3.51).

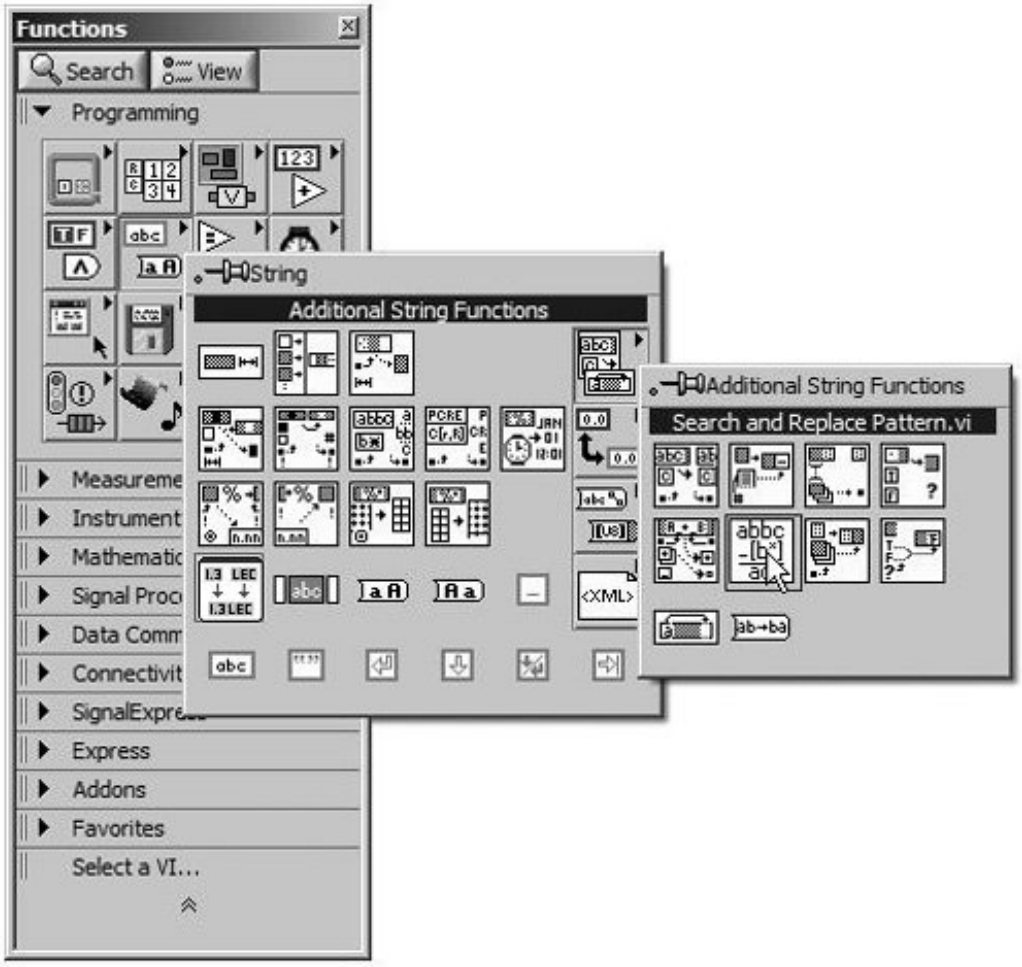
Figure 3.51. A floating subpalette created by right-clicking on a subpalette icon



Note that some subpalettes have subpalettes containing more objects; these are denoted by a little triangle in the upper-right corner of the icon and a raised appearance (see Figure 3.52). We'll discuss specific subpalettes and their objects in the next chapter.

Figure 3.52. Functions palette with two levels of floating subpalettes visible

[View full size image]



The Controls and Functions palettes can also be accessed by popping up in an empty area of the front panel or block diagram. "Popping up" is defined as right-mouse-clicking. (On Mac OS X you can also pop up by <control>-clicking.) You can also pop up with the soon-to-be-discussed Pop-up tool.

The Thumbtack: "Pinning" Down Palettes

If you use a subpalette frequently, you may want to "tear it off" by releasing the mouse button over the *thumbtack* located at the upper left of the palettethe palette becomes *pinned*, meaning that it will remain open as a floating window until you close it. This thumbtack is available when you navigate hierarchically through palettes by right-clicking (Windows and Linux) or control-clicking (Mac OS X),

as we just described. You now have a stand-alone window that you can position anywhere and then close when you're done with it. You can leave open as many subpalettes as you like.

When you pin a subpalette from a pop-up menu, it will be set to the Icon Palette View Format (which was described earlier) regardless of the Palette View Format of the palette it was created from.

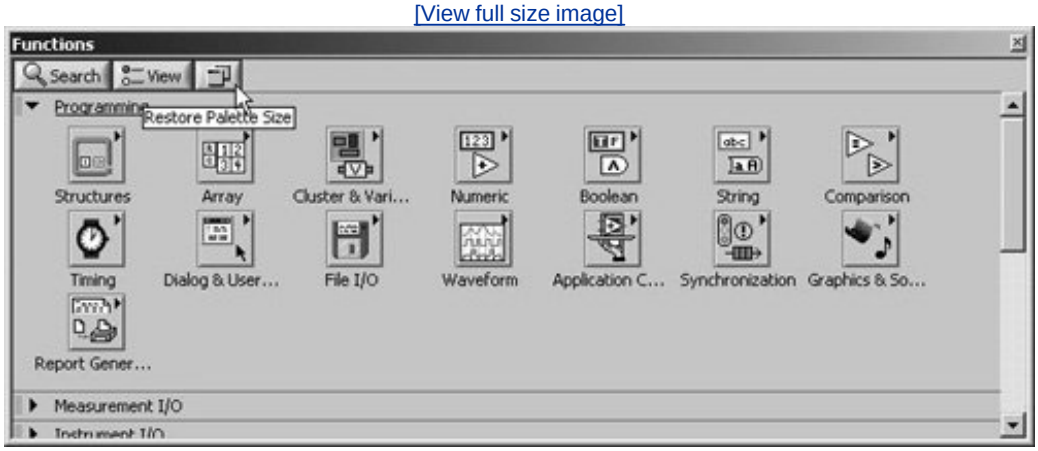
Resizing (and Restoring) Palettes

You can resize a palette window, once it has been *pinned* (as described previously) and is floating freely, to suit your needs. After resizing a palette window, the Restore Palette Size button will appear in the toolbar at the top of the window (see Figure 3.53). Pressing this button will cause the palette window to resize to the default.

Restore Palette Size Button

Figure 3.53. A resized palette and the Restore Palette Size toolbar button

[View full size image]



Customizing the Palettes

Because there are so many different objects on LabVIEW's numerous palettes, you may wish to customize your palettes. This section describes some of the ways you can do this.

Favorites

If you find that you are using certain functions or VIs often, you can make them more easily accessible by adding them to the Favorites category. You can add items to the Favorites category by right-clicking on the object and selecting Add Item to Favorites from the shortcut menu. You can remove items in the Favorites category, by right-clicking on the item within the Favorites category and selecting Remove Item from Favorites from the shortcut menu.

Editing the Palette

If LabVIEW's default organization of the Controls and Functions palettes doesn't fit your needs, you can customize them according to your whim. Access the palette editor by selecting Tools>>Advanced>>Edit Palette Set. . . from the menu. From here, you can customize the palettes by adding new subpalettes, hiding items, or moving them from one palette to another. For example, if you create a VI using trigonometric functions, you can place it in the existing Trigonometric subpalette for easy access. Editing the palettes is handy for placing your most frequently used functions at the top level for easy access and burying those pesky functions you never want to see again at the bottom of a subpalette.

User Libraries

If you have VIs that are shared across several projects, you can add them to the User Libraries palette category by placing them in the `user.lib` folder of the LabVIEW installation. LabVIEW will automatically add these VIs to the User Libraries category of the Functions palette. Similarly, custom controls placed in the `user.lib` folder will appear in the User Controls category of the Controls palette.

Tools Palette

A *tool* is a special operating mode of the mouse cursor. You use tools to perform specific editing and operation functions, similar to how you would use them in a standard paint program. The Tools palette (see Figure 3.54) is accessed from the View>>Tools Palette menu. From this palette you can select and configure tools.

Figure 3.54. Tools palette

Automatic Tool Selection allows LabVIEW to automatically choose the best tool from the Tools palette, depending on the location of your cursor relative to the object you are manipulating. You can disable automatic tool selection by clicking the Automatic Tool Selection button on the Tools palette, shown in Figure 3.54. Press the Automatic Tool Selection button on the Tools palette to enable automatic tool selection again. There is a LabVIEW option called Lock Automatic Tool Selection that locks your cursor to Automatic Tool Selection mode. If this option is enabled, then pressing the <Tab> key will not cycle through the various tools; rather it will enable the automatic tool selection again. If Lock Automatic Tool Selection is not enabled, then pressing the <Shift-Tab> key can be used enable automatic tool selection again after another tool was manually selected.

Automatic Tool Selection

The Operating tool lets you change values of front panel controls and indicators. You can operate knobs, switches, and other objects with the Operating toolhence the name. It is the only front panel tool available when your VI is running or in run mode (described shortly).

Operating Tool

The Positioning tool selects, moves, and resizes objects.

Positioning Tool

The Labeling tool creates and edits text labels.

Labeling Tool

The Wiring tool wires objects together on the block diagram. It is also used to assign controls and indicators on the front panel to terminals on the VI's connector.

Wiring Tool

The Color tool brightens objects and backgrounds by allowing you to choose from a multitude of hues. You can set both foreground and background colors by clicking on the appropriate color area in the Tools palette. If you pop up on an object with the Color tool, you can choose a hue from the color palette that appears.

Color Tool

The Pop-up tool opens an object's pop-up menu when you click on the object with it. You can use it to access pop-up menus instead of the standard method for popping up (right-clicking under Windows and Linux, and <control>-clicking on Mac OS X).

Pop-up Tool

The Scroll tool lets you scroll in the active window.

Scroll Tool

The Breakpoint tool sets breakpoints on VI diagrams to help you debug your code. It causes execution to suspend so you can see what is going on and change input values if you need to.

Breakpoint Tool

The Probe tool creates probes on wires so you can view the data traveling through them while your VI is running.

Probe Tool

Use the Color Copy tool to pick up a color from an existing object; then use the Color tool to paste that color onto other objects. This technique is very useful if you need to duplicate an exact shade but can't remember which one it was. You can also access the Color Copy tool when the Color tool is active by holding down the <control> key on Windows, <option> on Mac OS X, and <alt> on Linux.

Color Copy Tool

If the LabVIEW option Lock Automatic Tool Selection is not active, you can use the <tab> key to tab through the Tools palette instead of clicking on the appropriate tool button to access a particular tool. Or press the spacebar to toggle between the Operating tool and the Positioning tool when the panel window is active and between the Wiring tool and the Positioning tool when the diagram window is active. The <tab> and spacebar shortcuts cycle through the most frequently used tools for your conveniencetry using them, and see if they don't save you time!

You can also access a temporary copy of the Tools palette by pop-up clicking (<shift>-right click for Windows and Linux, and <control-shift>-click on Mac OS X).

Automatic Tool Selection

When Automatic Tool Selection is enabled, the cursor will appear as a cross with a small dot in the upper-right quadrant, as shown in Figure 3.55.

Figure 3.55. Mouse cursor with Automatic Tool Selection enabled

As you move the cursor over an object, it will change to the tool most appropriate for the job, based on the location of the cursor over the object being manipulated.

For example, if you move the cursor over the outer edge of numeric control, the cursor will become the Positioning tool (see Figure 3.56).

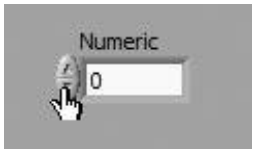
Positioning Tool

Figure 3.56. Automatic Tool Selection of the Positioning tool

If you move the cursor over the increment or decrement button of the numeric control, the cursor will become the Operating tool, allowing you to change the value of the numeric (see Figure 3.57).

Operating Tool

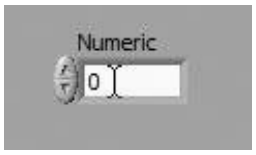
Figure 3.57. Automatic Tool Selection of the Operating tool



If you move the cursor over the text area of the numeric control, it will change to the Labeling tool, allowing you to edit the value using the keyboard (see Figure 3.58).

Labeling Tool

Figure 3.58. Automatic Tool Selection of the Labeling tool



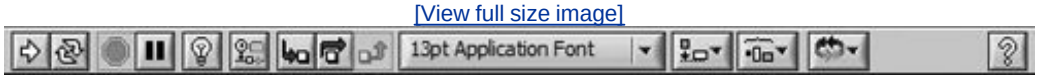
PREV NEXT

The Toolbar

The *Toolbar*, located at the top of LabVIEW windows, contains buttons you will use to control the execution of your VI, as well as text configuration options and commands to control the alignment and distribution of objects (see Figure 3.59). You'll notice that the Toolbar has a few more options in the block diagram than in the front panel, and that a few editing-related options disappear when you run your VI. If you're not sure what a button does, hold the cursor over it until a tip strip appears, describing its function.

Figure 3.59. Toolbar

[View full size image]



Run Button

Run Button (Active)

Run Button (Broken)

The Run button, which looks like an arrow, starts VI execution when you click on it. It changes appearance to *active* when a VI is actually running. When a VI won't compile, the run button appears *broken*.

Continuous Run Button

The Continuous Run button causes the VI to execute over and over until you hit the stop button. It's kind of like a GOTO statement (sort of a "programming no-no"), so use it sparingly.

Abort Button

The Abort button, easily recognizable because it looks like a tiny stop sign, becomes active when a VI begins to execute; otherwise, the Abort button is grayed out. You can click on this button to halt the VI.

Using the Abort button is like pulling the power cord on your computer. Your program will stop immediately rather than coming to a graceful end, and data integrity can be lost this way. You should always code a more appropriate stopping mechanism into your program, as we will demonstrate later.

Pause Button

The Pause button pauses the VI so that you can use single-step debugging options such as step into, step over, and step out. Hit the Pause button again to continue execution.

Step Into Button

Step Over Button

Step Out Button

The single-step buttonsStep Into, Step Over, and Step Outforce your VI to execute one step at a time so you can troubleshoot. We'll talk more about how to use them in Chapter 5, "Yet More Foundations."

Execution Highlight Button

The Execution Highlight button causes the VI to highlight the flow of data as it passes through the diagram. When execution highlight is on, you can see intermediate data values in your block diagram that would not otherwise appear.

Retain Wire Values

The Retain Wire Values button causes the VI's wires to store the value that flowed through them the last time the VI executed. This is very useful for debugging. You can view the value stored in a wire by placing a probe on the wire. The probe value will be set to the value stored in the wire.

Warning Button

The Warning button appears if you have configured your VI to show warnings and you have any warnings outstanding. You can list the warnings by clicking on the button. A warning is not an error; it just alerts you that you are doing something you may not have intended (for example, if there are objects on the block diagram that are hidden behind other objects and cannot be seen).

You can change the font, size, style, justification, and color of LabVIEW text from the Text Settings button on the Toolbar (see Figure 3.60).

Figure 3.60. Text Settings button

LabVIEW has an automatic alignment mechanism to help you line up and evenly space your icons. Select the objects you want to align by dragging around them with the Positioning tool, then go to the Align Objects button on the Toolbar and choose how you want to align them (top edges flush, left edges flush, vertical centers, etc.). If you want to set uniform spacing between objects, use the Distribute Objects button in a similar fashion (see Figure 3.61).

Figure 3.61. The Align Objects and Distribute Objects buttons

For resizing multiple objects to the same size, use the Resize Objects button (see Figure 3.62). You can resize to the maximum height and/or width, the minimum height and/or width, or you can explicitly set the width and height values using the Set Width and Height option (not shown in Figure 3.62) of the Resize Objects submenu.

Figure 3.62. Resize Objects button

In a similar fashion, LabVIEW lets you group objects together to treat them as one control for graphical editing purposes, as well as set the depth order (often called the *Z-order*) of objects, so that you can specify what objects should go in front of or behind others. You can do this with the Reorder Objects button (see Figure 3.63).

Figure 3.63. Reorder Objects button

Run Mode and Edit Mode

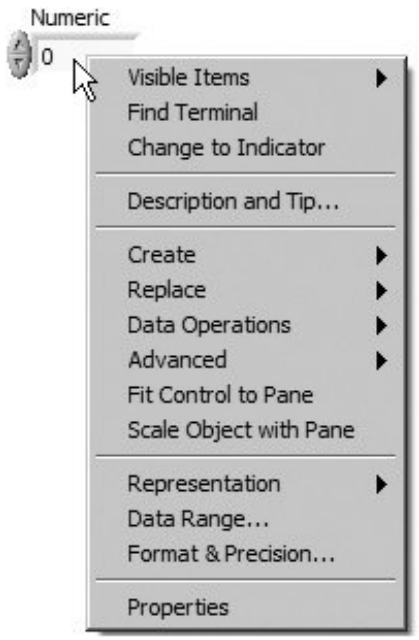
When you open a VI, it opens in *edit mode* so you can make changes to it. When you run a VI, it automatically goes into *run mode* and you can no longer edit. Only the Operating tool is available on the front panel when the VI is in run mode. When your VI completes execution, your VI reverts to edit mode (unless you manually switched it to run mode before you ran itthen it stays in run mode). You can switch to run mode by selecting Change to Run Mode from the Operate menu; switch to edit mode by choosing Change to Edit Mode. To draw a parallel with text-based languages, if a VI is in run mode, it has been successfully compiled and awaits your command to execute. Most of the time, you will not need to concern yourself with run and edit modes. But if you accidentally find that you suddenly have only the Operating tool, and you can't make any changes, at least now you'll know why.

If you prefer to open VIs in run mode (perhaps so uninvited users can't make changes), select Options. . . from the Tool menu. Go to the Miscellaneous options and choose "Open VIs in Run Mode."

Pop-Up Menus

As if pull-down menus didn't give you enough to learn about, we will now discuss the other type of LabVIEW menu: the pop-up menu. You will probably use pop-up menus more often than any other LabVIEW menu. To pop up, position the cursor over the object whose menu you desire; then click the right mouse button on Windows and Linux machines, or hold down the <control> key and click on the Mac. You can also click on the object with the Pop-up tool. A pop-up menu will appear (see Figure 3.64).

Figure 3.64. A pop-up menu



Virtually every LabVIEW object has a pop-up menu of options and commands. Options available in this pop-up menu depend on the kind of object, and they are different when the VI is in edit mode or run mode. For example, a numeric control will have a very different pop-up menu than a graph indicator. If you pop up on empty space in a front panel or block diagram, you will get the Controls or Functions palette, respectively.

You will find that instructions throughout this book guide you to select a command or option from an object pop-up menu, so try popping up now!

How to Pop Up

Windows and Linux: right mouse click on the object.

Mac: <command>-click on the object.

All Platforms: Click on the object with the Pop-up tool.

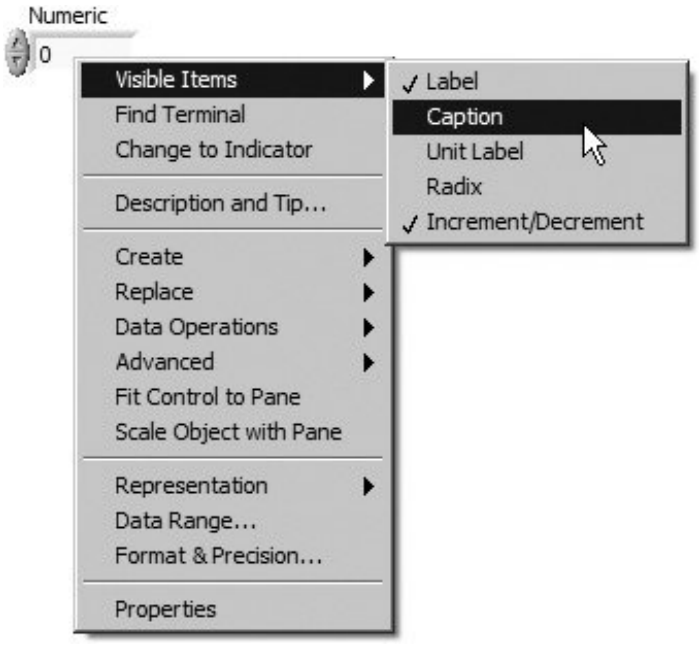
Pop-up menus are ever-present in LabVIEW. They contain most configuration options for an object. So remember, when in doubt about how to do something, try popping up!

If the Color tool is active, you will see a color palette when you pop up instead of the pop-up menu that appears when other tools are active.

Pop-Up Menu Features to Keep in Mind

Many pop-up menu items expand into submenus called hierarchical menus, denoted by a right arrowhead (see Figure 3.65).

Figure 3.65. A pop-up submenu



Hierarchical menus sometimes have a selection of mutually exclusive options. The currently selected option is denoted by a check mark for text-displayed options, or surrounded by a box for graphical options.

Some menu items pop up dialog boxes containing options for you to configure. Menu items leading to dialog boxes are denoted by ellipses (. . .).

Menu items without right arrowheads or ellipses are usually commands that execute immediately upon selection. A command usually appears in verb form, such as Change to Indicator. When selected, some commands are replaced in the menu by their inverse commands. For example, after you choose Change to Indicator, the menu selection becomes Change to Control.

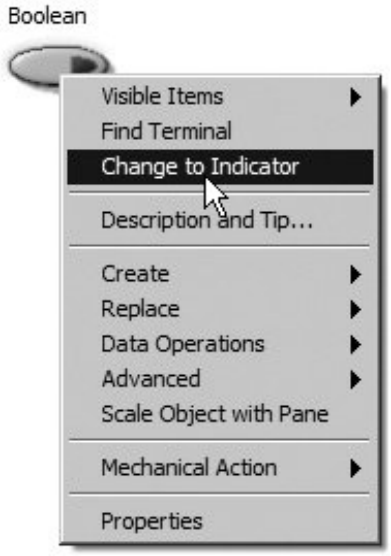
Sometimes different parts of an object have different pop-up menus. For example, if you pop up on an object's label, the menu contains only a Size to Text option. Popping up elsewhere on the object gives you a full menu of options. So if you pop up and don't see the menu you want, try popping up elsewhere on the object.

Pop-Up Features Described

Pop-up menus allow you to specify many traits of an object. The following options appear in

numerous pop-up menus (for example, the one shown in Figure 3.66), and we thought they were important enough to describe them individually. We'll let you figure out the other options, since we would put you to sleep detailing them all. Feel free to skim over this section and refer back when necessary.

Figure 3.66. Pop-up menu of a front panel control



Visible Items

Many items have Visible Items menus with which you can show or hide certain cosmetic features like labels, captions, scrollbars, or wiring terminals. If you select Visible Items, you will get another menu off to the side, listing options of what can be shown (this list varies depending on the object). If an option has a check next to it, that option is currently visible; if it has no check, it is hidden. Release the mouse on an option to toggle its status.

Find Terminal and Find Control/Indicator

If you select Find Terminal from a front panel pop-up menu, LabVIEW will locate and highlight its corresponding block diagram terminal. If you select Find Control/Indicator from a block diagram pop-up menu, LabVIEW will show you its corresponding front panel object. As a short-cut, you can simply double-click a control, indicator, or terminal to obtain the same effecthighlighting its block diagram or front panel counterpart. (But be aware that LabVIEW can be reconfigured to open a control or indicator in the Control Editor when double-clickedyou'll learn more about the Control Editor in Chapter 4.)

Change to Control and Change to Indicator

By selecting Change to Control, you can turn an existing control (an input object) into an indicator (an output object), or vice versa if you select Change to Indicator. When an object is a control, its pop-up menu contains the option to Change to Indicator. When it is an indicator, the pop-up menu reads Change to Control.

Because Change to Control/Indicator is an option in the pop-up menu, it is easy to accidentally select it without realizing what you've done. Controls and indicators are not functionally interchangeable in a block diagram, so the resulting errors may befuddle you.

A control terminal in the block diagram has a thicker border than an indicator terminal. Always pay attention to whether your objects are controls or indicators to avoid confusion!

Description and Tip

Selecting this option will allow you to enter a description and a "tip." The description will appear in the Help window for that control, and the tip will show up when you place the mouse cursor over this control (this type of help is sometimes called tool-tip or hesitation help).

Create

The Create option is an easy way for you to create a property node, local variable, or reference for a given object (these advanced topics will be covered in detail in Chapter 13).

Replace

The Replace option is extremely useful. It gives you access to the Controls or Functions palette (depending on whether you're in the front panel or block diagram) and allows you to replace the object you popped up on with one of your choice. Where possible, wires will remain intact.

Data Operations

The Data Operations pop-up menu has several handy options to let you manipulate the data in a control or indicator:

- Reinitialize to Default returns an object to its default value, while Make Current Value Default sets the default value to whatever data is currently there.

- Use Cut Data, Copy Data, and Paste Data to take data out or put data into a control or indicator.

Advanced

The Advanced pop-up option gives you access to less-frequently used features that let you fine-tune the appearance and behavior of the control or indicator:

- Key Navigation . . . Use "Key Navigation . . ." to associate a keyboard key combination with a front panel object. When a user enters that key combination while a VI is running, LabVIEW acts as if the user had clicked on that object, and the object becomes the key focus (*key focus* means the cursor is active in that field).
- Synchronous Display is a selectable option that forces LabVIEW to refresh the display of this control or indicator every time its value is changed on the block diagram (otherwise, LabVIEW will update the display at a regular interval, when the block diagram is not excessively busy). This option can add significant overhead, so you shouldn't use it unless you have a good reason.
- Customize. . . will bring up the *Control Editor* to allow you to customize the graphical appearance of the control. We'll talk about creating your own custom controls in Chapter 17, "The Art of LabVIEW Programming."
- Hide Control/Indicator. You can choose to hide a front panel object using this option, which comes in handy when you don't want the user to see the front panel object but still need it in the diagram. If you need to show the front panel object again, you must select "Show Control/Indicator" on the pop-up menu of the block diagram terminal.
- Enabled State allows you to set a control's state as enabled, disabled, or disabled & grayed. This comes in handy if you still want to show a control or indicator on the front panel, but you don't want the user to use it.

There are some other pop-up options that are specific to different types of controls (numeric, Boolean, etc.), but we'll leave them for later.

Don't worry about memorizing all of these features right nowyou'll come across them as you work with LabVIEW, and they'll make a lot more sense!

The same object will have a different pop-up menu in run mode than it will in edit mode. If you cannot find a certain pop-up option, it is either not present for that object, you need to switch modes, or you should pop up elsewhere on the object.

Properties

The Properties pop-up menu opens a dialog that allows editing of all the properties that are applicable for the object. Sometimes, other pop-up menu options will cause the Properties dialog to open at a specific tab of the Properties dialog. For example, the Format and Precision menu option of a numeric object will open the Format and Precision tab of the Numeric Properties dialog.

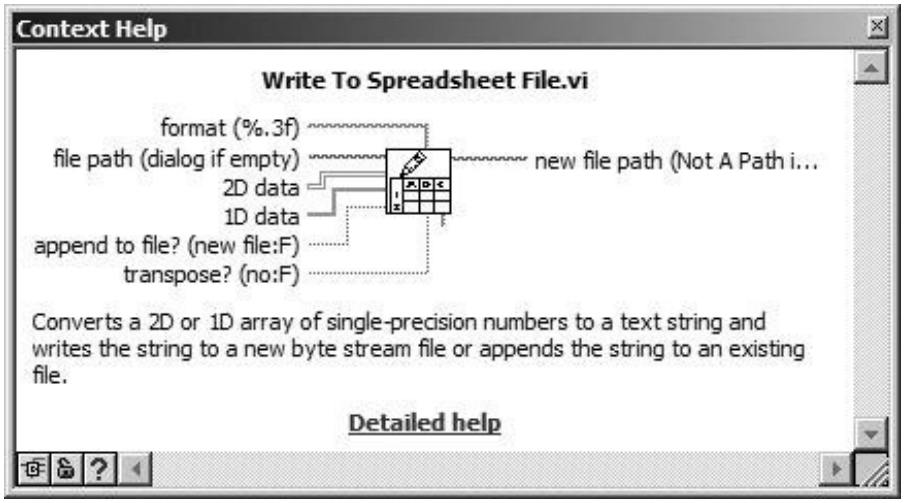
PREV NEXT

Help!

The Context Help Window

The LabVIEW *Context Help window* offers indispensable help information for functions, constants, subVIs, and controls and indicators. To display the window, choose Show Context Help from the Help menu or use the keyboard shortcut: <control-H> on Windows, <command-H> on the Mac, and <alt-H> on Linux. If your keyboard has a <help> key, you can press that instead. You can resize the Help window and move it anywhere on your screen to keep it out of the way (see Figure 3.67).

Figure 3.67. Help window



When you hold the cursor over a function, a subVI node, or a VI icon (including the icon of the VI you have open, at the top-right corner of the VI window), the Help window shows the icon for the function or subVI with wires of the appropriate data type attached to each terminal. *Input wires point to the left, and output wires point to the right*. Terminal names appear beside each wire. If the VI has a description associated with it, this description is also displayed.

Lock Button

For some subVIs or functions, the Help window will show the names of required inputs in bold, with default values shown in parentheses. In some cases, the default value can be used and you do not need to wire an input at all. You can lock the Help window so that its contents do not change when

you move the mouse by selecting Lock Context Help from the Help menu or by pressing the Lock button in the Help window.

If you position the Wiring tool over a specific node on a function or subVI, the Help window will flash the labeled corresponding node so that you can make sure you are wiring to the right place. Sometimes you may need to use the scrollbar to see all of the text in the Help window.

For VIs and functions with large numbers of inputs and outputs, the Help window can be overwhelming, so LabVIEW gives you the choice between simple or detailed views. You can use the simple view to emphasize the important connections and de-emphasize less commonly used connections.

Simple/Detailed Help Button

Switch between views by pressing the Simple/Detailed Help button on the lower-left corner of the Help window. In simple help view, required connections appear in bold text; recommended connections appear in plain text; and optional connections are not shown. Wire stubs appear in the place of inputs and outputs that are not displayed, to inform you that additional connections exist (and you can see them in the detailed help view).

In detailed help view, required connections appear in bold text; recommended connections appear in plain text; and optional connections appear as disabled text.

If a function input does not need to be wired, the default value often appears in parentheses next to the input name. If the function can accept multiple data types, the Help window shows the most common type.

Online Help

Online Help Button

LabVIEW's Help window provides a quick reference to functions, VIs, controls, and indicators. However, there are times when you'd prefer to look at a more detailed, indexed description for information on using a VI or function. LabVIEW has extensive online help that you can access by selecting Contents and Index . . . from the Help menu or by pressing the Online Help button in the Help window.

You can type in a keyword to search for, view an extensive keyword index, or choose from a variety of topics to browse through. You can also set your own links to online help documents, which we'll talk about in Chapter 17.

Currently, not all LabVIEW VIs link to online help; if this is the case, the online help menu item and online help button will be grayed out.

PREV NEXT

Express VIs

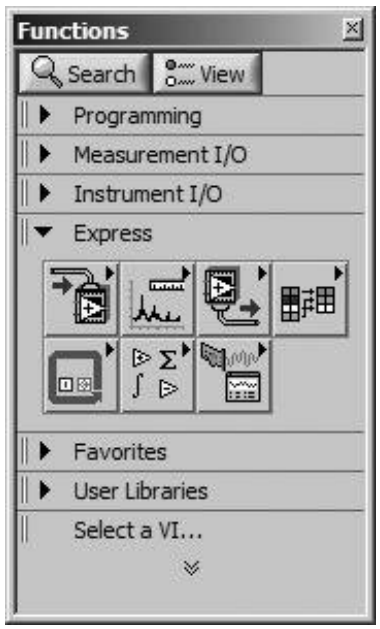
Express VIs are a special kind of LabVIEW function. They are different from "normal" LabVIEW functions because, among other things, they will allow you to define the behavior of the function using a wizard or configuration dialog. They are designed to help you quickly perform test, measurement, and analysis tasks. In essence, whenever you put an Express VI on your block diagram, LabVIEW immediately and automatically builds lower-level LabVIEW code for you, "behind the scenes." Some Express VI functions include user dialogs, timing functions, signal simulation, formula calculation, file reading, and much more. Think of the Express VIs as a library of solutions for programming problems that are common to many applications. These solutions save you the work of having to write the low-level code yourself. LabVIEW writes the LabVIEW code for you. Pretty cool!

Because Express VIs are different than normal functions in LabVIEW, we'll use the Express icon to indicate places in this book that refer to Express VIs.

Whenever you see this icon, you'll know that the section is going to talk about an Express VI.

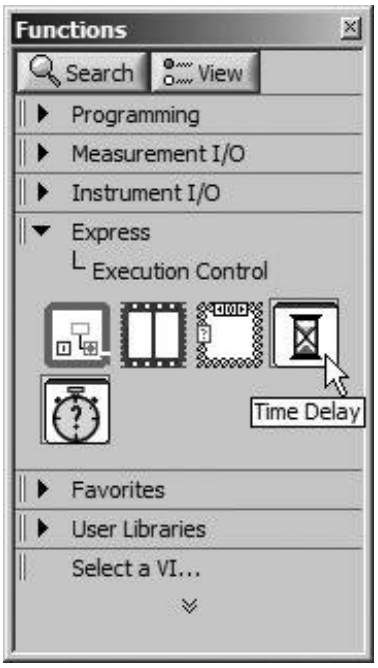
You can find many Express VIs in the Express category of the Functions palette, as shown in Figure 3.68.

Figure 3.68. Express category of the Functions palette



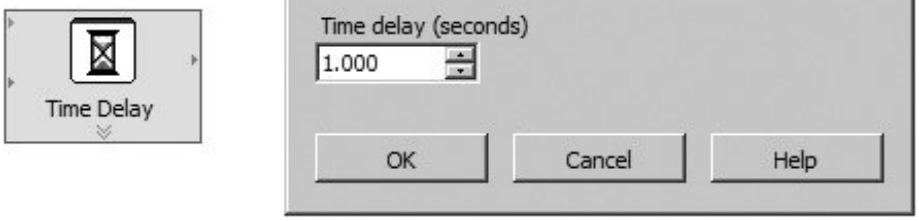
When you place an Express VI onto a block diagram, you will see a dialog box open that allows you to configure the Express VI's behavior. You can also open this configuration dialog box by double-clicking on the Express VI, or by right-clicking on it and choosing Properties from the pop-up menu. Let's try this by placing the Time Delay Express VI, which is found in the Express>>Execution Control subpalette of the Functions palette, onto the block diagram of a VI (see Figure 3.69).

Figure 3.69. Time Delay Express VI, found on the Express>>Execution Control palette



Immediately after placing this Express VI onto the block diagram, a configuration dialog opens (shown in Figure 3.70), allowing us to configure the Time Delay (seconds) value.

Figure 3.70. Time Delay Express VI on the block diagram with configuration dialog open



After we press the OK button, the Time Delay (seconds) value will be applied to the Express VI.

If you place the cursor over the Express VI (with the Context Help window open), then the configuration parameter values will be displayed in the Context Help window. This allows you to quickly inspect the configuration of the Express VI without having to open its configuration dialog.

Express VIs appear on the block diagram as expandable nodes, with icons surrounded by a blue background. We will learn more about expandable nodes in the next section of this chapter.

If the Express VI's configuration dialog box does not open automatically after it is placed on the block diagram, it is because the LabVIEW option Configure Express VIs immediately is not checked. This option is found in the Block Diagram section of the Tools>>Options . . . menu dialog.

You can convert an Express VI into a regular subVI, by right-clicking on it and choosing Open Front Panel from the pop-up menu. You will be presented with a confirmation dialog asking if you want to convert the Express VI into a standard subVI.

After you have converted an Express VI to a standard subVI, you will not be able to access the configuration dialog again and you will have to edit any configuration parameters by modifying the subVI or wiring new values to its inputs.

PREV NEXT

Displaying SubVIs as Expandable Nodes

SubVIs may be viewed as either icons or expandable nodes. Express VIs are viewed as expandable nodes, by default. To view a subVI as an expandable node, right-click on it and uncheck the View As Icon option by selecting it from the pop-up menu, as shown in Figure 3.71. This will cause the subVI's appearance to change, such that its icon is surrounded by a yellow background, as shown in Figure 3.72.

Figure 3.71. SubVI with View As Icon setting enabled, as also indicated in its visible pop-up menu

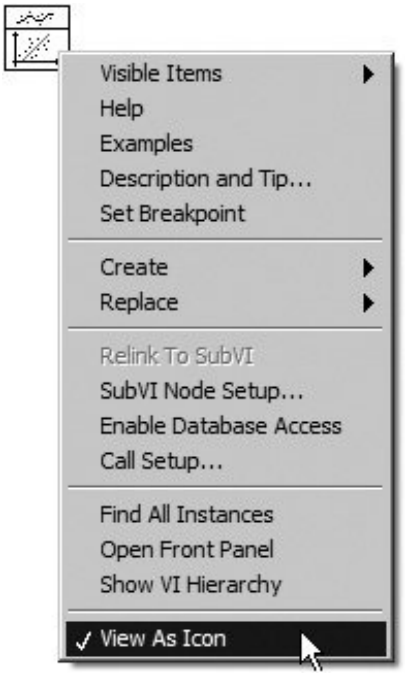
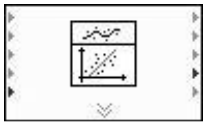
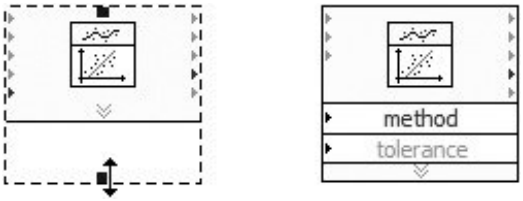


Figure 3.72. SubVI with View As Icon setting disabled, appearing as an expandable node



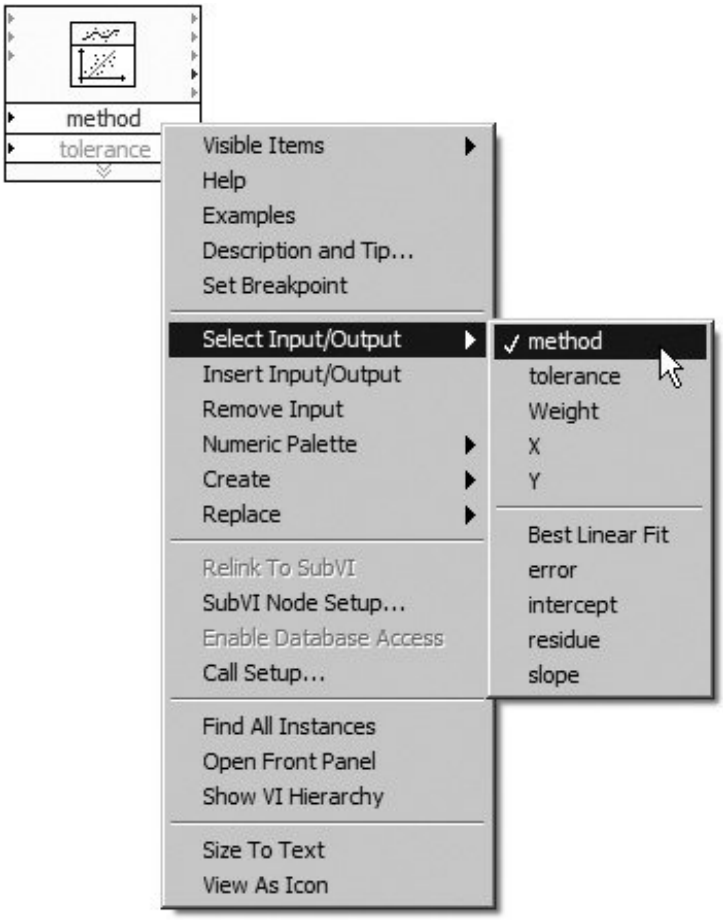
We can resize the subVI vertically, which causes inputs and outputs to appear below the subVI's icon, as shown in Figure 3.73. Note that inputs and outputs made available are no longer available to the left and right of the subVI's icon.

Figure 3.73. Resizing (expanding) a subVI configured as an expandable node



You can also show and hide inputs and outputs from the subVI's pop-up menu, using the Select Input/Output submenu to change an input/output, Insert Input/Output to insert a new input/output, and Remove Input/Output to remove an input/output (see Figure 3.74).

Figure 3.74. Selecting the visible inputs and outputs of a subVI configured as an expandable, from its pop-up menu



You can make subVIs automatically appear as expandable nodes, by checking the LabVIEW option Place subVIs as expandable, which is found in the Block Diagram section of the Tools>>Options. . . menu dialog.

PREV NEXT

A Word About SubVIs

If you want to take full advantage of LabVIEW's abilities, you must understand and use the hierarchical nature of the VI. A *subVI* is simply a stand-alone program that is used by another program. After you create a VI, you can use it as a subVI in the block diagram of a higher-level VI as long as you give it an icon and define its connector. A LabVIEW subVI is analogous to a subroutine in C or another text-based language. 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 (memory permitting, of course).

If a block diagram has a large number of icons, you can group them into a subVI to maintain the simplicity of the block diagram. You can also use one subVI to accomplish a function common to several different top-level VIs. This modular approach makes applications easy to debug, understand, and modify. We'll talk more about how to build subVIs later, but it's such an important part of the LabVIEW programming environment that we want you to keep it in mind as you're learning the basics.

LabVIEW does not let a VI call itself as a subVI (a technique known as recursion). LabVIEW also does not let a VI call subVIs that use it as a subVI (at any level of the VI hierarchy). If you attempt to place a subVI that would violate this rule, LabVIEW will display a dialog stating that recursive use of the VI is not allowed.

There is an advanced technique that uses VI Server and the Call By Reference Node to allow a reentrant VI to call itself dynamically (recursively). You can learn about VI Server and the Call By Reference node in Chapter 15, "Advanced LabVIEW Features."

Activity 3-2: Front Panel and Block Diagram Basics

In this activity, you will practice some simple exercises to get a feel for the LabVIEW environment. Try to do the following basic things on your own. If you have any trouble, glance back through the chapter for clues.

1. Open a new VI and toggle between the front panel and block diagram.

Use the keyboard shortcuts listed in the pull-down menus!

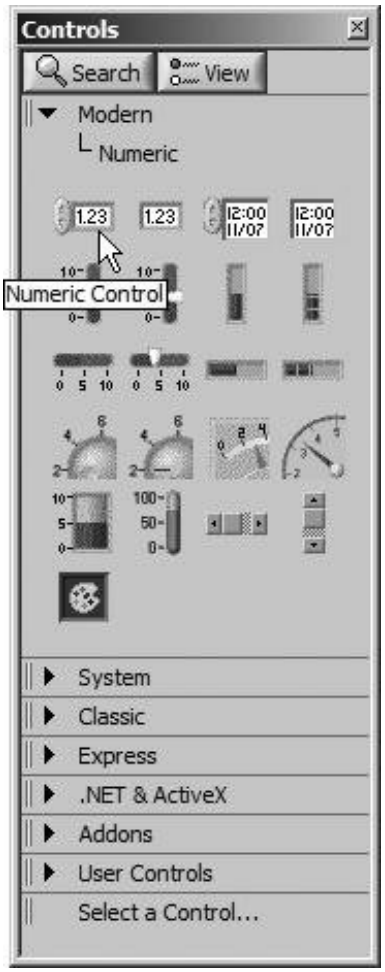
2. Resize the windows so that both front panel and block diagram are visible simultaneously. You may need to move them around.

Do this using the standard resizing technique for your platform. Another hint: Try the Tile function!

3. Drop a numeric control, a string control, and a Boolean indicator on the front panel by selecting them from the Controls palette.

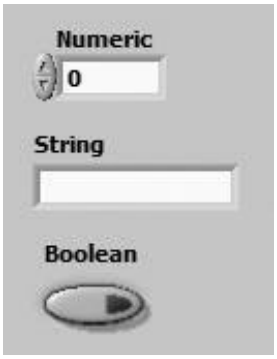
   To get the numeric control, click on the Numeric button in the Modern category of the Controls palette and select Numeric Control from the subpalette that appears (see Figure 3.75).

Figure 3.75. The Modern>>Numeric palette showing the Numeric Control as well as other numeric controls and indicators



Now click your mouse on the front panel in the location where you want your digital control to appear. Voilàthere it is! Now create the string control and Boolean indicator in the same fashion (see Figure 3.76).

Figure 3.76. Your front panel after adding numeric, string, and Boolean controls



Notice how LabVIEW creates corresponding terminals on the block diagram when you create a front panel object. Also notice that floating-point numeric terminals are orange (integer numerics will be blue), strings are pink, and Booleans are green. This color-coding makes it easier for you to distinguish between data types.

4. Now pop up on the digital numeric control (by right-mouse-clicking on Windows and Linux or <command>-clicking on Mac) and select Change to Indicator from the pop-up menu. Notice how the appearance of the numeric changes on the front panel (the little up and down arrows go away). Also notice how the terminal on the block diagram changes (the border is much thinner for indicators). Switch the object back and forth between control and indicator until you can easily recognize the differences on both front panel and block diagram. Note that for some objects (like a few Booleans), front panel indicators and controls can look the same, but their block diagram terminals will always be different.

5. Choose the Positioning tool from the floating Tools palette, and then select an object on the front panel. Hit the <delete> key to remove it. Delete all front panel objects so you have an empty front panel and block diagram.

Positioning Tool

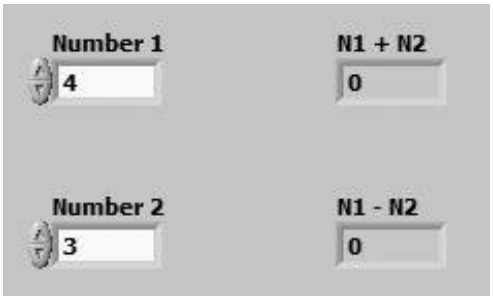
6. Drop another digital control from the Numeric subpalette of the Controls palette onto the front panel. If you don't click on anything first, you should see a little box above the control. Type `Number 1`, and you will see this text appear in the box. Click the Enter button on the Toolbar to enter the text. You have just created a label. Copy the digital control, `Number 1`, to the clipboard by selecting it with the Positioning tool and then selecting Edit>>Copy from the menu. Then de-select `Number 1` by clicking on an empty location of the front panel. Paste a copy of `Number 1` onto the front panel by selecting Edit>>Paste from the menu. Note that LabVIEW has auto-incremented the name of the new control to `Number 2`. Create a digital indicator labeled `N1+N2` and a digital indicator labeled `N1-N2`.

Enter Button

Operating Tool

Use the Operating tool to click on the increment arrow of `Number 1` until it contains the value "4.00." Give `Number 2` a value of "3.00" (see Figure 3.77).

Figure 3.77. Your front panel after adding the numeric controls and indicators and setting the control values



7. Switch back to the diagram. Drop an Add function from the Numeric subpalette of the Functions palette in the block diagram (this works just like creating front panel objects). Now repeat the process and drop a Subtract function.

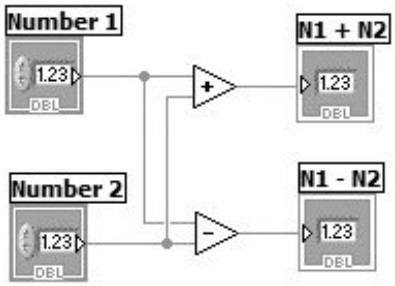
8. Pop up on the Add function and select the Visible Items>>Terminals option (you'll notice that before you select it, the option is not checked, indicating that terminals are not currently shown). Once you show them, observe how the input and output terminals are arranged; then redisplay the standard icon by again selecting Visible Items>>Terminals (this time, the option appears with a checkmark next to it, indicating that terminals are currently shown).

9. Bring up the Help window by using either the keyboard shortcut or the Show Context Help command from the Help menu. Position the cursor over the Add function. The Help window provides valuable information about the function's use and wiring pattern. Now move the cursor over the Subtract function and watch the Help window change.

10. You may have to use the Positioning tool to reposition some of the terminals, as shown in Figure 3.78. Then use the Wiring tool to wire the terminals together. First select it from the Tools palette, and then click once on the numeric control's terminal and once on the appropriate terminal on the Add function to draw a wire. A solid orange line should appear. If you mess up and get a dashed black line instead of a solid orange one, select the wire fragment with the Positioning tool and hit the <delete> key; then try again. Click once and release to start the wire, click any time you want to tack a new segment (which turns a corner), and click on a destination to finish the wire.

Wiring Tool

Figure 3.78. Your block diagram after positioning the terminals and functions and wiring everything up



Notice that when you pass the Wiring tool over the Add and Subtract functions, little wire stubs appear, showing where the terminals are located. In addition, as you pass the cursor over a terminal, its name appears in a tip strip. Like learning to type, wiring can be kind of tricky until you get the hang of it, so don't worry if it feels a little awkward right now!

11. Switch back to the front panel and pop up on the icon pane (the little window in the upper-right corner). Select Show Connector from the menu (see Figure 3.79). Observe the connector that appears. If you can't get the pop-up menu to appear, you are probably trying to pop up in the icon pane of the block diagram.

Figure 3.79. The Show Connector option, accessible from the pop-up menu of the VI's icon pane

Now pop up again on the connector and look at its menu to see the configuration options you have. The connector defines the input and output parameters of a VI so that you can use it as a subVI and pass data to it. You can choose different patterns for your connectors depending on how many parameters you need to pass.

As you learned in the earlier section, "SubVIs, the Icon, and the Connector," it is usually best to use the default connector (having 12 terminals) so that you have extra terminals, should you add controls or indicators to your subVI at a later time.

Show the icon again by selecting Show Icon. Remember, the icon is just the pictorial representation of a VI; when you use a VI as a subVI, you will wire to this icon in the block diagram of the top-level VI just like you wired to the Add function.

12. Run the VI by clicking on the Run button. The N1+N2 indicator should display a value of "7.00" and N1-N2 should be "1.00." Feel free to change the input values and run it over and over.


Run Button (Active)

13. Save the VI by selecting Save from the File menu. Call it Add.vi and place it in your MYWORK directory.

Congratulations! You have now mastered several important basic LabVIEW skills!

PREV NEXT

Wrap It Up!

The LabVIEW environment has three main parts: the *front panel, block diagram*, and the *icon/connector*. The front panel is the user interface of the programyou can input data through *controls* and observe output data through *indicators*. When you place an object on the front panel using the Controls palette, a corresponding terminal appears in the block diagram, making the front panel data available for use by the program. Wires carry data between *nodes*, which are LabVIEW program execution elements. A node will execute only when all input data is available to it, a principle called *dataflow*.

A VI should also have an *icon* and a *connector*. When you use a VI as a subVI, its icon represents it in the block diagram of the VI you use it in. Its connector, usually hidden under the icon, defines the input and output parameters of the subVI.

LabVIEW has two types of menus: pull-down and pop-up. *Pull-down* menus are located in the usual menu spot at the top of your window or screen, while *pop-up* menus can be accessed by "popping up" on an object. To pop up, right-mouse click on Windows and Linux machines and <command>-click on the Mac, or click with the Pop-up tool. Pull-down menus tend to have more universal commands, while pop-up menu commands affect only the object you pop up on. Remember, when in doubt about how to do something, pop up to see its menu options!

The Tools palette gives you access to the special operating modes of the mouse cursor. You use these tools to perform specific editing and operation functions, similar to how you would use them in a standard paint program. You will find front panel control and indicator graphics located in the Controls palette, and block diagram constants, functions, and structures in the Functions palette. These palettes often have objects nestled several layers down in *subpalettes*, so make sure you don't give up your search for an object too soon.

The Help window provides priceless information about functions and how to wire them up; you can access it from the Help menu. LabVIEW also contains extensive online help that you can call up from the Help menu or by pressing the online help button in the Help window. Between these two features, your questions should never go unanswered!

You can easily turn any VI into a subVI by creating its icon and connector and placing it in the block diagram of another VI. Completely stand-alone and modular, subVIs offer many advantages: they facilitate debugging, allow many VIs to call the same function without duplicating code, and offer an alternative to huge messy diagrams.

Don't worry if this seems like a lot to remember. It will all become natural to you as you work your way through the book.

4. LabVIEW Foundations

Overview

Get ready to learn about LabVIEW's basic principles in this chapter. You will learn how to use different data types and how to build, change, wire, and run your own VIs. You will also learn some helpful shortcuts to speed your development. Make sure you understand these fundamentals before you proceed, because they are integral to all developments you will achieve in LabVIEW.

Goals

- Become comfortable with LabVIEW's editing techniques
- Learn the different types of controls and indicators, and the special options available for each
- Master the basics of creating a VI, such as wiring and editing
- Create and run a simple VI

Key Terms

- Options
- Numeric
- String
- Boolean
- Path
- Ring control
- Format and precision
- Numeric representation
- Bad wires
- Shortcuts
- Label
- Caption

Creating VIs: It's Your Turn Now!

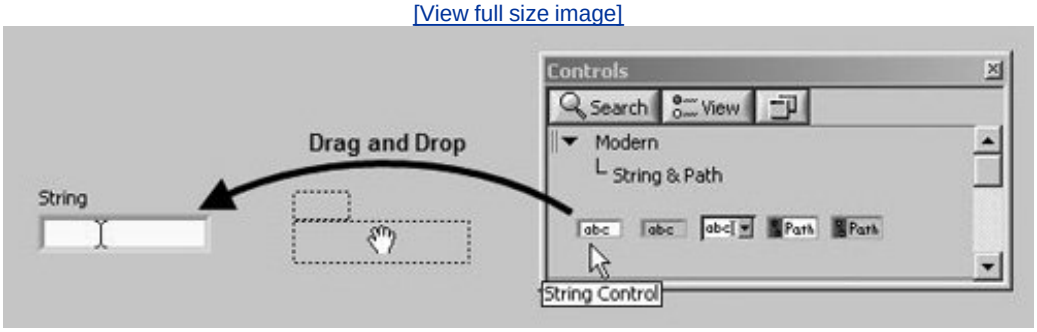
We've gone over a few basics of the LabVIEW environment, and now we're going to show you exactly how to build your own VIs. Because people remember things better if they actually do them, you might step through these instructions on your computer as you read them so that you can learn the techniques more quickly.

Placing Items on the Front Panel

You will usually want to start your programs by "dropping" controls and indicators on a front panel to define your user inputs and program outputs. You've done this once or twice before in activities, but we'll mention it here for reference (and as a good way to start this interactive section). As you run the cursor over the Controls palette, you will see the names of subpalettes and items appear in the mouse pointer's tip strip (as shown by the "String Control" text in Figure 4.1). Left-clicking on subpalette icons will navigate the palette tree into that subpalette.

Figure 4.1. Placing an item on the front panel by dragging it from the Controls palette and dropping it on the front panel

[View full size image]



From the palette, you can drag and drop controls, indicators, and decorations onto your front panel, as shown in Figure 4.1.

You can also access the Controls palette by popping up in an empty area of the front panel.

Now create a new VI and drop a digital control on your front panel.

Remember, when you drop an item on the front panel, its corresponding terminal appears on the block diagram. You might find it helpful to select Tile Left and Right from the Windows menu so that you can see the front panel and block diagram windows at the same time.

Labeling Items

Labels are blocks of text that are the names of the specific components on front panels and block diagrams. An object first appears in the front panel window with a default name for a label (e.g., "Numeric," "String," etc.). Until you click somewhere else with the mouse, the label text will be selected and you can rename the label by entering text from the keyboard. If you click somewhere else with the mouse first, the default label will remain. After you enter text into a label, any one of the following actions completes the entry:

- Press <enter> on the numeric keypad.
- Click on the enter button in the Tools palette.
- Click somewhere outside the label on the front panel or block diagram.
- Press <shift-enter> or <shift-return>.

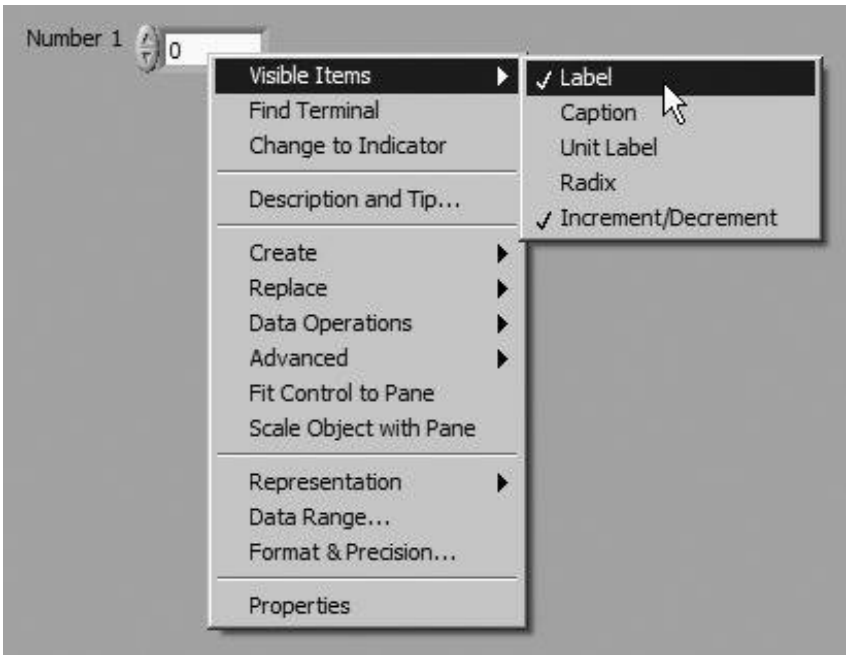
The label appears on the front panel object's corresponding block diagram terminal as well as the front panel object.

LabVIEW has two kinds of labels: owned labels and free labels. Owned labels belong to and move with a particular object; they annotate that object only. When you create a control or indicator on the front panel, a default owned label accompanies it, awaiting input. A front panel object and corresponding block diagram terminal will have the same owned label. A free label is not associated with any particular object and can be created and deleted at will on either the front panel or block diagram.

Owned "Object" Labels

You can select Visible Items>> *Label* from the owning object's pop-up menu to create or change a label that isn't currently visible (see Figure 4.2). You can hide owned labels, but you cannot copy or delete them independently of their owners. Structures and functions on the block diagram come with a default label that is hidden until you show it. You may want to edit this label to reflect the object's function in your programthis is an excellent way to document your code.

Figure 4.2. The Visible Items>> Label pop-up menu option



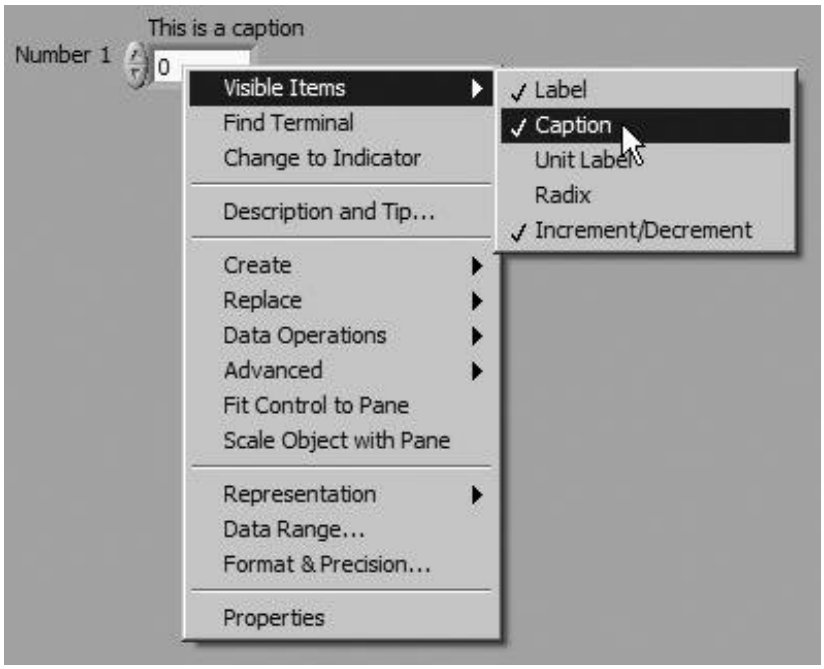
You can also show labels of subVIs (which are really just their names), but you cannot edit them.

Now label the digital control you just created `Number 1.` You may have to select Visible Items>>*Label* if the label is empty.

Captions

In addition to labels, front panel objects can have a *caption*. A caption is very much like a labelsome text that describes the control or indicator. To enter a caption for a control or indicator, pop up on it and select Visible Items>>Caption (see Figure 4.3).

Figure 4.3. The Visible Items>>Caption pop-up menu option



So why would you need or want a caption *and* a label? Most of the time, you probably don't. In certain advanced programming situations (which we'll touch on later), you need to use a control's label to reference it in the software, and you may want to have a separate caption that is more user friendly. Generally, you will want to keep your label short and concise; for example, using "Temperature" as a control's label but using a longer user-readable description for the caption ("This displays the current temperature in degrees Celsius"). Think of captions as a convenient way to link a comment to a front panel object. Also, it is worth mentioning that the caption can be changed programmatically (while your VI is running), but the label cannot.

Captions are a great way to localize your applicationdisplaying all the application's text in the specific language a user has selected. Using captions would allow you to programmatically change the language of all your front panel text!

For the following practice activities in LabVIEW, it will be helpful if you have the Tools palette visible and accessible (see Figure 4.4). We'll be asking you to select different tools (cursors) from the palette.

Figure 4.4. Tools palette

If you don't see the Tools palette in LabVIEW, go to the View menu and select "Tools Palette."

Creating Free Labels

Free labels are not attached to any object, and you can create, move, or dispose of them independently. Use them to annotate your panels and diagrams. Use the Labeling tool to create free labels and to edit virtually any visible text.

Labeling Tool

To create a free label, select the Labeling tool from the Tools palette and click anywhere in empty space. A small, bordered box appears with a text cursor at the left margin ready to accept typed input. Type the text you want to appear in the label and enter it in one of the four ways previously described. If you do not type any text in the label, the label disappears as soon as you click somewhere else. Create a free label on the front panel that says `hippopotamus.`

Free labels are a great way to document your block diagram, which is very important. And this is one of the grading criteria for the LabVIEW Developer Certification exams. Generally speaking, you can't have too much documentation. However, don't be literal (don't state the obvious; for example, "1 +

1 = 2" is just silly), but do convey the intent, or purpose, of your code (for example, "add noise onto simulated signal").

Changing Font, Style, Size, and Color of Text

You can change text attributes in LabVIEW using the options in the Font ring on the Toolbar. Select objects with the Positioning tool or highlight text with the Labeling or Operating tools, then make a selection from the Text Settings ring (see Figure 4.5). The changes apply to everything selected or highlighted. If nothing is selected, the changes apply to the default font and will affect future instances of text.

Figure 4.5. Text Settings ring

Change your `hippopotamus` label so that it uses 18-point font.

If you select Font Dialog . . . from the menu, a dialog box appears; you can change multiple font attributes at the same time using this dialog box.

LabVIEW uses System, Application, and Dialog fonts for specific portions of its interface. These fonts are predefined by LabVIEW, and changes to them affect all controls that use them.

- The Application font is the default font, used for the Controls palette, the Functions palette, and text in new controls.
- The System font is used for menus.
- LabVIEW uses the Dialog font for text in dialog boxes, and for Controls and Indicators found in the System subpalette of the Controls palette.

Placing Items on the Block Diagram

A user interface isn't much good if there's no program to support it. You create the actual program by placing functions, subVIs, and structures on the block diagram. To do this, access the Functions palette just like you did the Controls palette. Then select the item you want from a subpalette, and click on the diagram to place it.

Drop an Add function from the Programming>>Numeric subpalette of the Functions palette onto the block diagram.

Editing Techniques

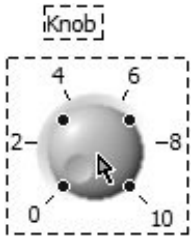
Once you have objects in your windows, you will want to be able to move them around, copy them, delete them, and so on. Read on to learn how.

Selecting Objects

Positioning Tool

You must select an item before you can move it. To select something, click the mouse button while the Positioning tool is on the object. When you select an object, LabVIEW surrounds it with a moving dotted outline called a *marquee*, shown in Figure 4.6.

Figure 4.6. Marquee



To select more than one object, <shift>-click on each additional object. You can also deselect a selected object by <shift>-clicking on it.

Another way to select single or multiple objects is to drag a selection rectangle around them. To do this, click in an open area with the Positioning tool and drag diagonally until all the objects you want to select lie within or are touched by the selection rectangle that appears. When you release the mouse button, the selection rectangle disappears and a marquee surrounds each selected object. The marquee is sometimes referred to as "marching ants," for obvious reasons. Once you have selected the desired objects, you can move, copy, or delete them at will.

You cannot select a front panel object and a block diagram object at the same time. However, you can select multiple objects on the same front panel or block diagram.

Clicking on an unselected object or clicking in an open area de-selects everything currently selected. <shift>-clicking on an object selects or deselects it without affecting other selected objects.

Now select the digital control you created earlier.

Moving Objects

You can move an object by selecting and dragging it to the desired location. If you hold down the <shift> key and then drag an object, LabVIEW restricts the direction of movement horizontally or

vertically (depending on which direction you first move the object). You can also move selected objects in small, precise increments by pressing the appropriate arrow key; hold down the <shift> key at the same time to make the arrow keys move objects by a larger amount.

If you change your mind about moving an object while you are dragging it, drag the cursor outside all open windows and the dotted outline will disappear. Then release the mouse button, and the object will remain in its original location. You can also press the Escape <esc> key to cancel a Move operation. While moving objects, if you move the object directly over the window edge, the window will auto-scroll. If you press the <shift> key while the window is auto-scrolling, it will move faster.

Move your digital control to the other side of the screen.

Duplicating Objects

Positioning Tool

You can duplicate LabVIEW objects after you have selected them. From the Edit menu, select the Copy option, click the cursor where you want the new object, and then select the Paste option. You can also duplicate an object by using the Positioning tool to <control>-click on the object if you use Windows, <option>-click if you use Mac OS X, and <alt>-click on Linux machines. Then drag the cursor away while still holding down the mouse. You will drag away the new copy, displayed as a dotted line, while the original stays in place. You can also duplicate front panel and block diagram objects from one VI to another. For example, if you select a block of code on one diagram and drag it to another, the appropriate wires will stay in place and any necessary front panel objects will be created.

Unless you have set the Tools>>Options setting Block Diagram>>Delete/copy panel terminals from the diagram to TRUE, you cannot duplicate control and indicator terminals on the block diagramyou must copy the items on the front panel.

Copy your digital control using both methods. You should now have three digital controls labeled `Number 1`, `Number 2`, and `Number 3` on your front panel and three corresponding terminals on the block diagram. Notice how LabVIEW automatically changes the label numbers for you. To find out which one belongs to which, pop up on a control or on a terminal and select Find Terminal or Find Control (or simply double-click on it). LabVIEW will find and highlight the object's counterpart.

Deleting Objects

To delete an object, select it and then choose Delete from the Edit menu or just press <delete>.

Unless you have set the Tools>>Options setting Block Diagram>>Delete/copy panel terminals from the diagram to TRUE, you can only delete controls and indicators from the front panel. If you try to delete their terminals on the block diagram, the deletions are ignored.

Although you can delete most objects, you cannot delete control or indicator components such as labels and digital displays. Instead, you can hide these components by selecting Visible Items from the pop-up menu and then deselecting the appropriate option.

Delete one of your digital controls.

Resizing Objects

Positioning Tool

You can change the size of most objects. When you move the Positioning tool over a resizable object, resizing handles appear at the corners and sides of the object, as in Figure 4.7.

Figure 4.7. Resizing handles

When you pass the Positioning tool over a resizing handle, the cursor changes to the Resizing tool. Click and drag this cursor until the dotted border outlines the size you want (see Figure 4.8).

Figure 4.8. Using the resizing handles to resize an object

To cancel a resizing operation, press the Escape <esc> key or continue dragging the frame corner outside the window until the dotted frame disappears. Then release the mouse button. The object maintains its original size.

If you hold down the <shift> key while you resize, the object will change size only horizontally, vertically, or in the same proportions in both directions, depending on the object and in which direction you drag first.

If you hold down the <ctrl> key in Windows, the <option> key in Mac OS X, or the <meta> key in Linux, while changing an object's size, it will resize equally from both sides, or from all four sides if you resize a corner.

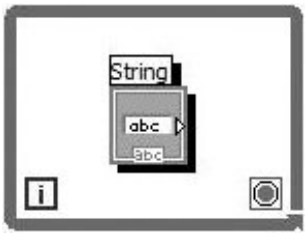
Resize one of your digital controls.

Some controls and indicators don't allow certain sizing operations. For example, digital controls can only grow horizontally (however, you can use a larger font to make the object bigger horizontally and vertically).

Moving, Grouping, and Locking Objects

Objects can sit on top of and often hide other objects, either because you placed them there or through some wicked twist of fate. LabVIEW has several commands in the Edit menu that move them relative to each other. You may find these commands very useful for finding "lost" objects in your programs. If you see an object surrounded by a shadow, chances are it's sitting on top of something. In Figure 4.9, the string control is not actually inside the loop; it is sitting on it.

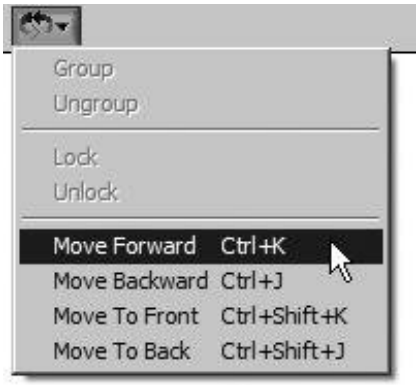
Figure 4.9. A terminal sitting (floating) on top of a While Loop, but not actually inside it



You can use the following options found in the Reorder ring (shown in Figure 4.10) to reorder objects:

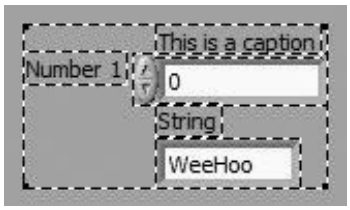
- Move To Front moves the selected object to the top of a stack of objects.
- Move Forward moves the selected object one position higher in the stack.
- Move To Back and Move Backward work similarly to Move To Front and Move Forward except that they move items down the stack rather than up.

Figure 4.10. The Reorder ring menu options



On the front panel, you can also group two or more objects together (see Figure 4.11). You do this by selecting the objects you want to group, and choosing Group from the Reorder ring. Grouping objects will make them behave as one object when you move them, resize them, or delete them.

Figure 4.11. Two controls that are grouped



Ungroup will "break up" the group back into the individual objects.

Lock objects will fix an object's size and position so it cannot be resized, moved, or deleted. This is handy if you are editing a front panel with a lot of objects and don't want to accidentally edit certain controls.

Coloring Objects

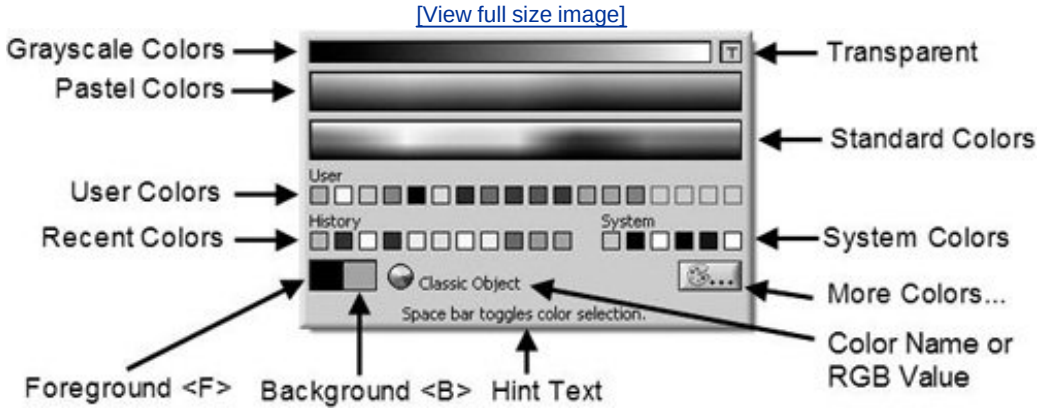
You can change the color of most LabVIEW objects, including controls, indicators, the front panel background, labels, and some block diagram elements. Not all elements can have their color changed. For example, block diagram terminals of front panel objects and wires use color codes for the type of data they carry, so you cannot change them.

Color Tool

To change the color of an object or the background window, pop up on it with the Color tool. The palette in Figure 4.12 appears in color.

Figure 4.12. Color tool pop-up window

[View full size image]



As you move through the palette while depressing the mouse button, the object or background you

are coloring redraws with the color currently touched by the cursor. This gives you a preview of the object in the new color. If you release the mouse button on a color, the object retains the selected color. To cancel the coloring operation, move the cursor out of the color palette before releasing the mouse button. Selecting the More Colors . . . button () on the palette dialog calls up another dialog box for picking custom colors and specifying exact RGB or HSL color values.

Many objects have both a foreground and a background color. The Color tool pop-up window allows you to specify both the Foreground and Background colorsthe selected colors are shown in the Foreground and Background color boxes in the lower-left corner of the window. You can choose which color you are adjusting by toggling between Foreground and Background using the spacebar key, or by pressing the <F> or <B> key for Foreground or Background (respectively).

Color one of your digital controls by popping up and selecting the color. Then color another control using the Tools palette method.

Matching Colors

Color Copy Tool

Sometimes it's hard to match a shade you've used, so you can also duplicate the color of one object and transfer it to a second object without going through the color palette. You can use the Color Copy tool on the Tools palette, which looks like an eye dropper (some call it the "sucker" tool), to set the active colors. Simply click with it on an object displaying the colors you want to pick up, and then switch to the Color tool to color other things.

You can also access the Color Copy tool by <control>-clicking under Windows, <option>-clicking on Mac OS X, and <alt>-clicking on Linux machines with the Color tool on the object whose color you want to duplicate. Then you can release the keystroke and click on another object with the Color tool; that object assumes the color you chose.

Transparency

If you select the box with a "T" in it from the color palette (located in the box in the upper-right corner) and color an item, LabVIEW makes the object transparent. You can use this feature to layer

objects. For instance, you can place invisible controls on top of indicators, or you can create numeric controls without the standard three-dimensional container. Transparency affects only the appearance of an object. The object responds to mouse and key operations as usual. Some objects cannot be made transparent, such as a front panel or block diagram. The box that usually contains the "T" will be replaced with an "X" if transparency is not allowed for the selected object.

Object Alignment and Distribution

Sometimes you want to make your VIs look just perfect, and you need a way to evenly line up and space your objects. LabVIEW's alignment and distribution functions make this easy. To align objects, select them with the Positioning tool (it's usually easiest just to drag a rectangle around them all, rather than <shift>-clicking on each one individually), then go to the Align ring, located in the Toolbar right next to the Text Settings ring, and choose how you want them lined up (see Figure 4.13). The Distribute ring works similarly to space objects evenly (see Figure 4.14).

Figure 4.13. Align ring

Figure 4.14. Distribute ring

Be careful when you use these functions, because sometimes you'll end up with all of your objects on top of each other and you'll wonder what happened. For example, if you have three buttons in a row, and you align by left edges, all left edges will be flush, and all the objects will be stacked on top of each other. If this happens, hit <ctrl>-Z to undo, or use the Positioning tool to pick them off one by one.

PREV NEXT

Activity 4-1: Editing Practice

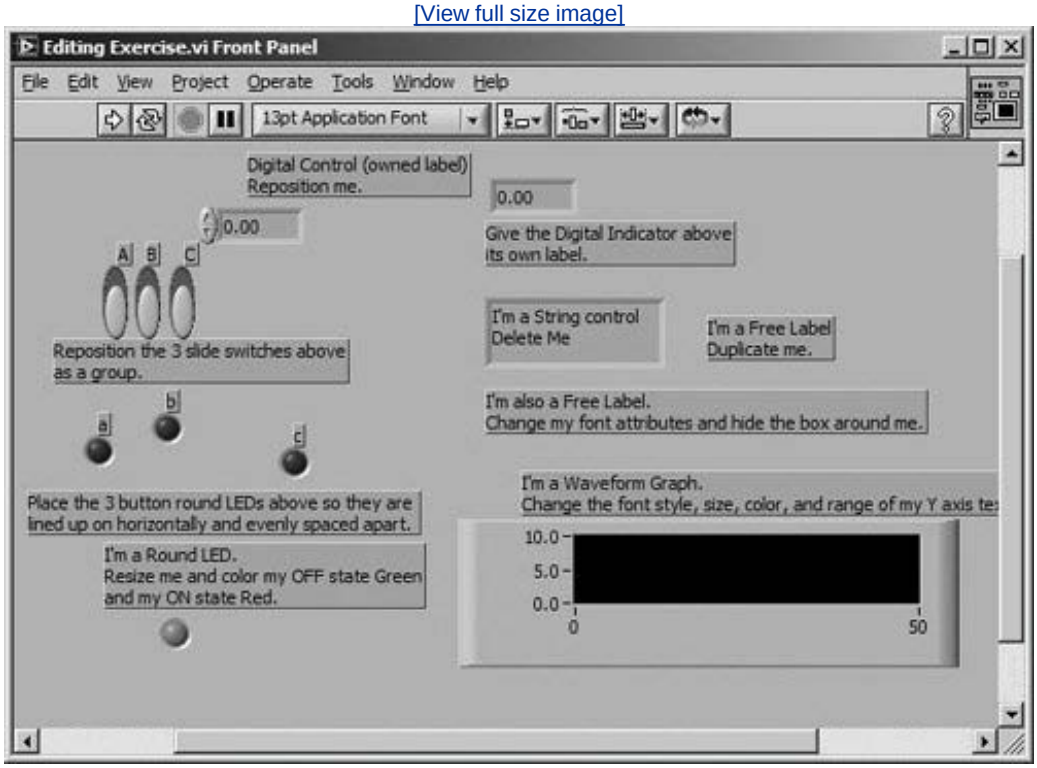
In this activity, you will practice some of the editing techniques you've just learned. Remember, just as the Controls palette is visible only when the front panel window is active, the Functions palette can only be seen when the block diagram window is up.

1. Open Editing Exercise.vi, found in the `EVERYONE\CH4` directory or folder. 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 shown in Figure 4.15.

Positioning Tool

Figure 4.15. Editing Exercise.vi front panel, which you will edit during this activity

[View full size image]



2. First, you will reposition the digital control. Using the Positioning tool, click on the digital control and drag it to another location. Notice that the label follows the controlthe control owns the label. Now, click on a blank space on the panel to deselect the control; then click on the label and drag it to another location. Notice that the control does not follow. An owned label can be positioned anywhere relative to the control, but when the control moves, the label will follow.

3. Reposition the three slide switches as a group. Using the Positioning tool, click in an open area near the three switches, hold down the mouse button, and drag until all the switches lie within the selection rectangle. Click on the selected switches and drag them to a different location.

4. Delete the string control by selecting it with the Positioning tool, and then pressing <delete> or selecting Delete from the Edit menu.

5. Duplicate the free label. Hold down the <control> key on a computer running Windows, the <option> key on Mac OS X, or the <alt> key under Linux; then click on the free label and drag the duplicate to a new location.


Labeling Tool

6. Change the font style of the free label. Select the text by using the Labeling tool. You can double-click on the text, or click and drag the cursor across the text to select it. Modify the selected text using the options from the Text Settings ring. Then hide the box around the label by popping up on the box with the Color tool and selecting the "T" (for transparent) from the color palette. Remember, use the right mouse button on Windows, Sun, and Linux and <command>-click on Macintosh to pop up on an item. Or you can use the Pop-up tool from the Tools palette and simply click on the object to access its pop-up menu.


Color Tool


Pop-up Tool

7. Now use the Font ring again to change the font style, size, and color of the Y-axis text on the Waveform Graph.


Enter Button

8. Create an owned label for the digital indicator. Pop up on the digital indicator by clicking the right mouse button under Windows, Sun, and Linux, or by clicking the mouse button while holding down the <command> key on the Mac; then choose Visible Items>>Label from the

pop-up menu. Type `Digital Indicator` inside the bordered box. Press <enter> on the numeric keypad, click the enter button on the Toolbar, or click the mouse button outside the label to enter the text.


Positioning Tool

9. 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 you resize.


Color Tool

10. Change the color of the round LED. Using the Color tool, pop up on the LED. While continuing to depress the mouse button, choose a color from the selection palette. When you release the mouse button, the object assumes the last color you selected. Now click with the Operating tool on the LED to change its state to ON, and then color the new state.


Vertical Centers Axis

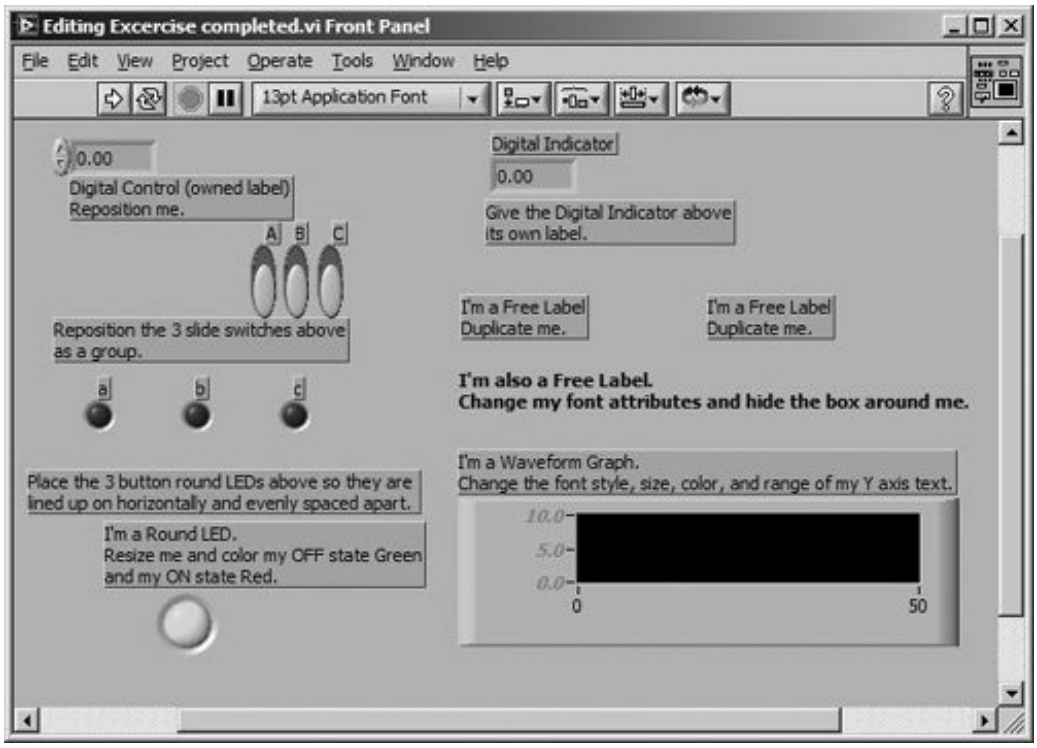
11. Place the three LED indicators so they are aligned horizontally and evenly spaced. Using the Positioning tool, click in an open area near the LEDs and drag a rectangle around them. Align them horizontally by choosing the Vertical Centers axis from the Align ring in the Toolbar. Then space the LEDs evenly by choosing Horizontal Centers axis from the Distribute ring.


Horizontal Centers Axis

12. Your panel should now look something like the one shown in Figure 4.16.

Figure 4.16. Editing Exercise.vi front panel, after you have completed this activity

[View full size image]



13. Close the VI by selecting Close from the File menu. Do not save any changes. Pat yourself on the backyou've mastered LabVIEW's basic editing techniques!

PREV NEXT

Basic Controls and Indicators and the Fun Stuff They Do

We're now going to talk a bit about the goodies contained in palettes of the Controls palette. LabVIEW has four types of "simple" controls and indicators: *numeric*, *Boolean*, *string*, and *paths*. You will also encounter a few more complex data types such as arrays, clusters, tables, charts, and graphs that we will expand on later.

The Controls palette is visible only when the front panel window is active, NOT when the block diagram is up. You can drive yourself crazy looking for it if you don't keep this in mind.

When you need to enter numeric or text values into any controls or indicators, you can use the Operating or Labeling tool. New or changed text is not registered until you press the <enter> key on the numeric keypad, click the enter button on the Toolbar, or click outside the object to terminate the editing session.

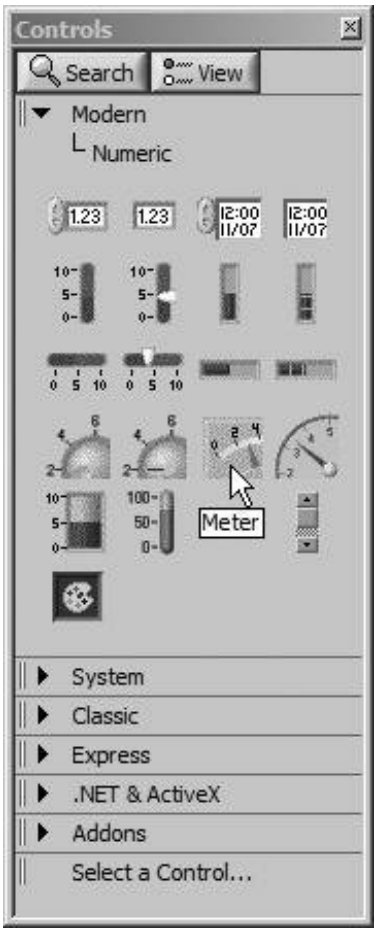
Hitting the <enter> key (Windows and Linux) or <return> key (Mac OS X) on the alphanumeric keyboard (NOT the one on the numeric keypad) enters a carriage return and does NOT register your change (unless you've configured your system otherwise, by checking the End text entry with Enter key option in the LabVIEW Options window, under the Environment options category). You must use the <enter> key on the numeric keypad to enter text to LabVIEW. If you must use the alphanumeric keyboard, hit <ctrl-enter> (Windows), <alt-enter> (Linux), or <option-return> (Mac OS X) to enter text.

Numeric Controls and Indicators

Numeric controls allow you to enter numeric values into your VIs; numeric indicators display numeric values you wish to see. LabVIEW has many types of numeric objects: knobs, slides, tanks,

thermometers, and, of course, the simple digital display. To use numerics, select them from the Modern>>Numeric subpalette of the Controls palette (see Figure 4.17). All numerics can be either controls or indicators, although each type defaults to one or the other. For example, a thermometer defaults to an indicator because you will most likely use it as one. By contrast, a knob appears on the front panel as a control because knobs are usually input devices.

Figure 4.17. Modern>>Numeric subpalette of the Controls palette



Representation

The appearance of numeric terminals on the block diagram depends on the representation of the data. The different representations provide alternative methods of storing data, to help use memory more effectively. Different numeric representations may use a different number of bytes of memory to store data, or may view data as *signed* (having the capacity for negative values) or *unsigned* (having only zero or positive values). Block diagram terminals are blue for integer data and orange for floating-point data (integer data has no digits to the right of the decimal point). The terminals contain a few letters describing the data type, such as "DBL" for double-precision floating-point data.

The numeric data representations available in LabVIEW are shown in Table 4.1, along with their size in bytes and a picture of a digital control terminal with that representation in the View As Icon setting (which is available from the terminal's pop-up menu) selected and not selected.

Table 4.1. Numeric Representations

Representation	*Abbreviation*	*Terminal (Icon)*	*Terminal*	*Size (bytes)*
byte	I8			1
unsigned byte	U8			1
word	I16			2
unsigned word	U16			2
long	I32			4
unsigned long	U32			4
quad	I64			8
unsigned quad	U64			8
single precision	SGL			4
double precision	DBL			8
extended precision	EXT			10[a]/12[b]/16[c]
complex single	CSG			8
complex double	CDB			16

Representation	*Abbreviation*	*Terminal (Icon)*	*Terminal*	*Size (bytes)*
complex extended	CXT			20[a]/24[b]/32[c]

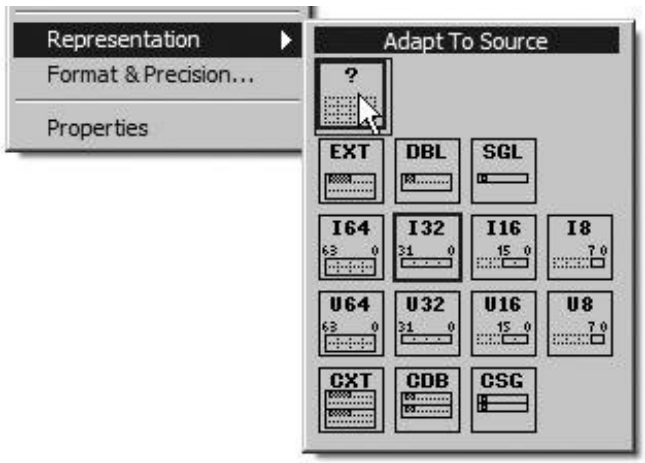
[a] Windows

[b] Mac OS X

[c] Linux

You can change the representation of numeric constants, controls, and indicators by popping up on the object and selecting Representation>>. Remember, you pop up on a numeric control or indicator by right-mouse-clicking. You can then choose from the palette shown in Figure 4.18.

Figure 4.18. The Representation pop-up submenu of numeric datatypes



If you are concerned about memory requirements, you will want to use the smallest representation that will hold your data without losing information, especially if you are using larger structures like arrays. Adapt To Source automatically assigns the representation of the source data to your indicatora good habit to get into. LabVIEW also contains functions that convert one data type to another, which will be covered in detail in Chapter 9, "Exploring Strings and File I/O," and in Chapter 14, "Advanced LabVIEW Data Concepts."

Format and Precision

LabVIEW lets you select whether your digital displays are formatted for numeric values or for time and date. If numeric, you can choose whether the notation is floating point, scientific, or engineering. If time, you can choose absolute or relative time in seconds. You can also choose the *precision* of the

display, which refers to the number of digits to the right of the decimal point, from 0 through 20. The precision affects only the display of the value; the internal accuracy still depends on the representation.

You can specify the format and precision by selecting Format & Precision . . . from an object's pop-up menu. The dialog box shown in Figure 4.19 appears. If you'd rather show time and date, choose Absolute Time from the listbox and your dialog box will change accordingly (see Figure 4.20).

Figure 4.19. Format and Precision tab of the Numeric Properties dialog

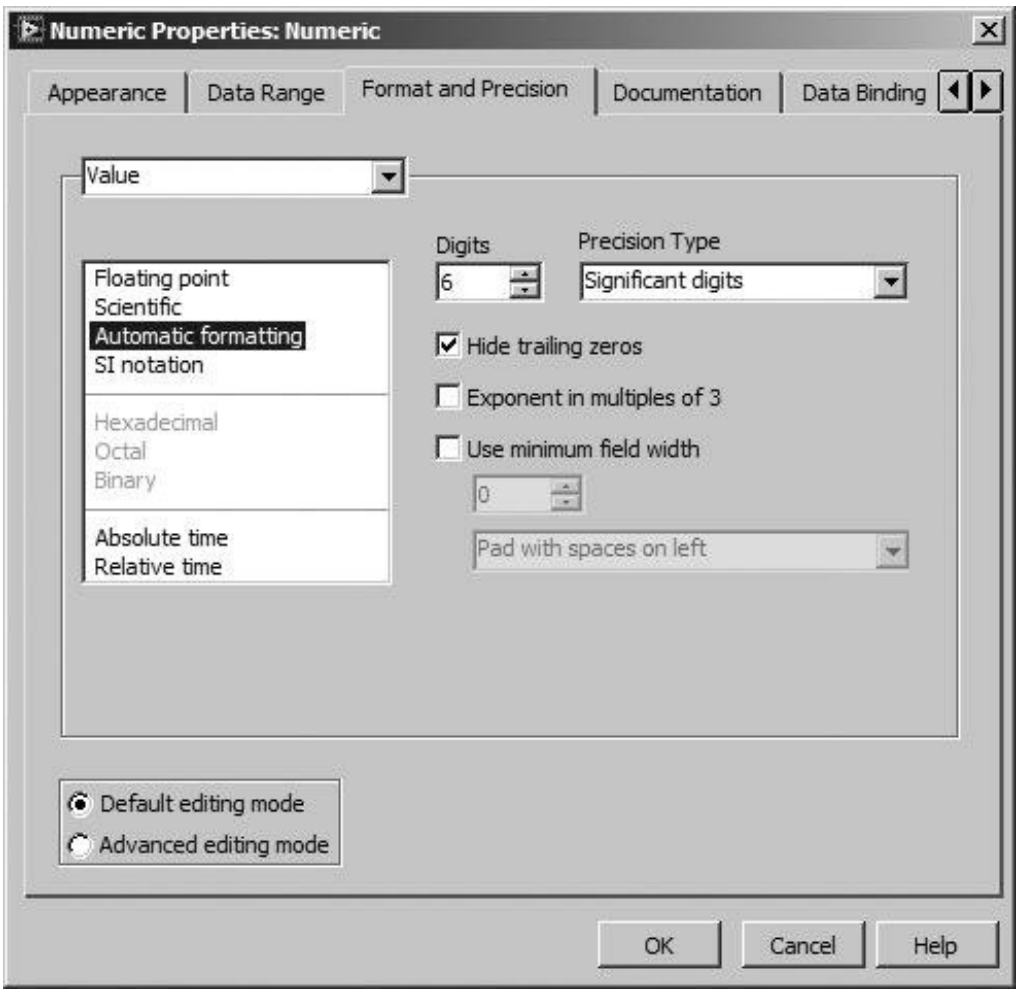
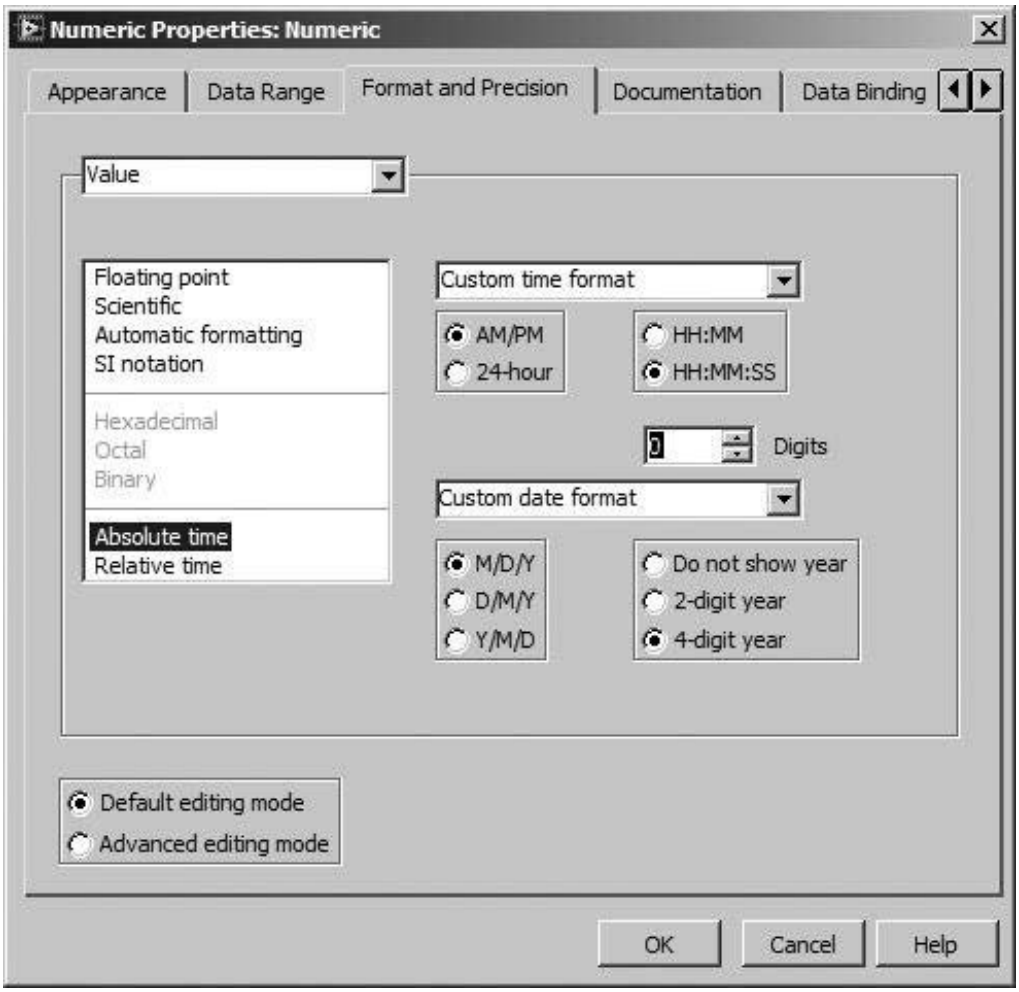


Figure 4.20. Configuring a numeric to display absolute time

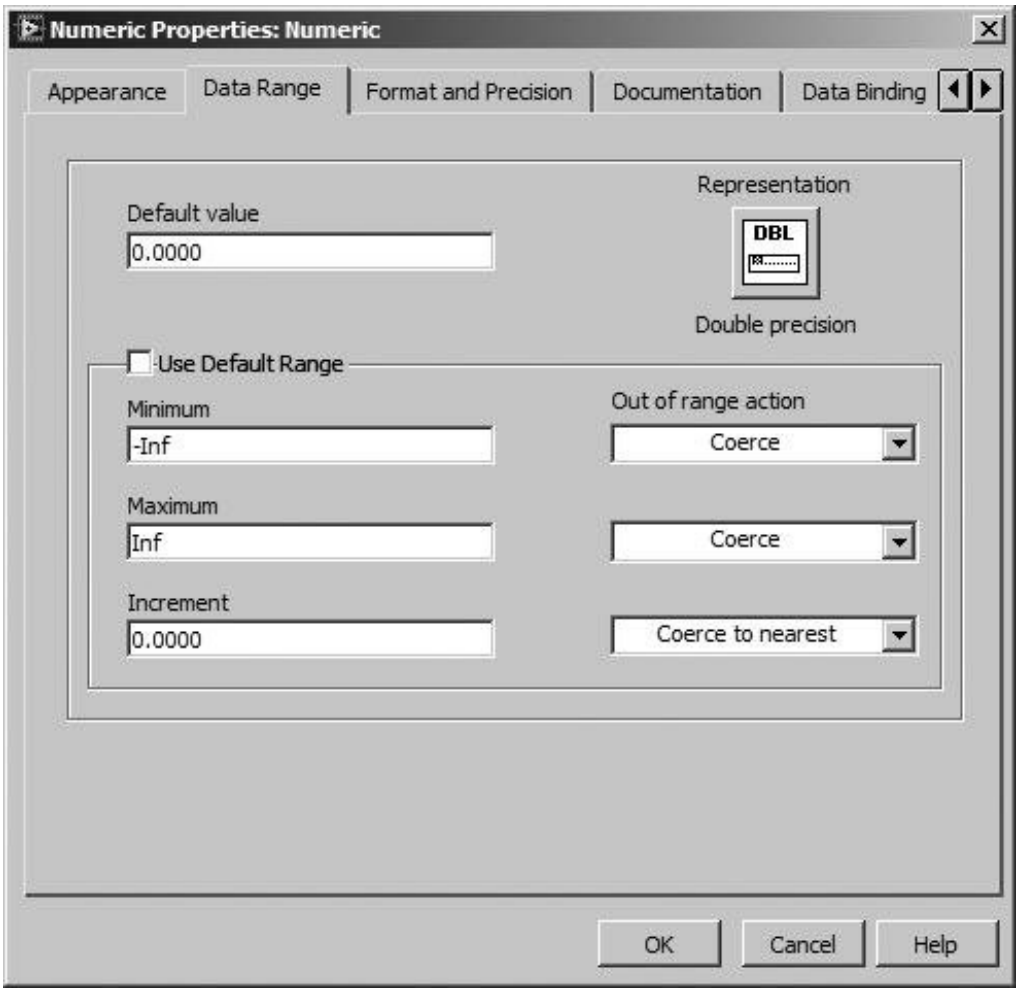


Numeric Range Checking

LabVIEW gives you the option to enforce user input data to a certain valid range of numeric values and increments of data. For example, you might only want to allow an input between zero and 100, in increments of two. You can set range checking by popping up on the appropriate numeric value and selecting Data Range . . . .

From the dialog box that appears (see Figure 4.21), you can leave the default representation (by leaving the box checked), or you can change numeric representation, input maximum and minimum acceptable values, set the increments you want, and change the default value for that object, as well as select a course of action to follow if values are out of range.

Figure 4.21. Data Range tab of the Numeric Properties dialog



- If you choose to Ignore out-of-range values, LabVIEW does not change or flag them. Clicking on the increment or decrement arrows of a control will change the value by the increment you set, up to the maximum values (or down to the minimum). However, you can still type in or pass a parameter out of the limits.

- If you choose to Coerce your data, LabVIEW will set all values you type in below the minimum to equal the minimum and all values above the maximum to equal the maximum. Values in improper increments will be rounded.

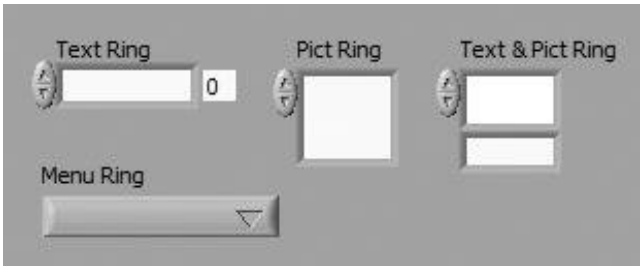
Numeric range and increment checking is only applicable to user input into front panel controls. It has no effect on values that are passed into a subVI's terminals (controls that have been wired to the subVI's connector pane)in this case, coercion does not occur and the values are passed into the controls, unchanged.

Rings

Rings are special numeric objects that associate unsigned 16-bit integers with strings, pictures, or both. You can find them in the Modern>>Ring & Enum and Classic>>Classic Ring & Enum subpalettes of the Controls palette. They are particularly useful for selecting mutually exclusive options such as modes of operation, calculator function, and so on.

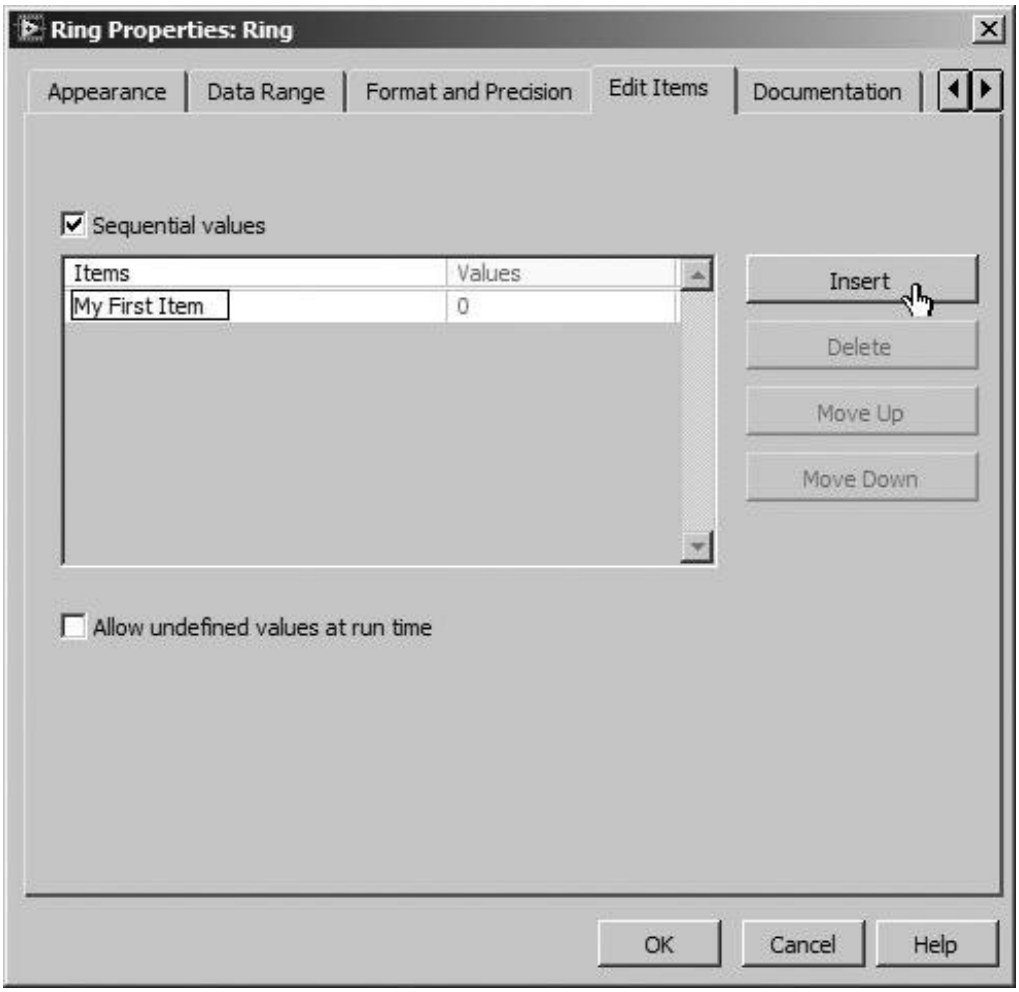
When you create a ring, you enter text or paste a picture into the ring that becomes associated with a certain number (zero for the first text message, one for the next, and so on). You can see this number (shown in Figure 4.22) by selecting Visible Items>>Digital Display from the ring's pop-up menu.

Figure 4.22. Ring controls



A new ring contains one item with a value of zero and an empty display. If you want to edit the items in the ring, select Edit Items . . . from the pop-up menu and a dialog will appear, as shown in Figure 4.23.

Figure 4.23. Edit Items tab of the Ring Properties dialog



The "*Items*" in the previous Ring Properties dialog are the text lines that will show up in the ring as the possible options to the user.

The "*Values*" are the numeric values that the block diagram will use for each item.

By default, LabVIEW creates sequential values for you (0, 1, 2, 3 . . .) for each item, so normally you do not need to worry about or edit this property.

If you click on a ring with the Operating tool, you will see a list of all possible messages or pictures, with the current one checked, as shown in Figure 4.24.

Figure 4.24. The list of ring items, shown after clicking on the ring with the Operating tool

Rings are useful if you want a user to select an option that will then correspond to a numeric value in the block diagram. Try dropping a ring on the front panel; then show the digital display and add a few items.

When adding items into a Ring, press <shift-enter> (Windows and Linux) or <shift-return> (Mac OS X) to complete the entry of the current item and add another black ring item after the item just entered. The key focus will remain in the new ring item, so you can quickly enter the new item. This is an extremely fast way to enter multiple sequential items. To quickly edit non-sequential items, use the Edit Items option from the pop-up menu, as previously described.

Booleans

Booleans are named for George Boole, an English logician and mathematician whose work forms the basis for Boolean algebra. For our purposes, you can think of Boolean as just a fancy word for "on or off." Boolean data can have one of two states: true or false. LabVIEW provides a myriad of switches, LEDs, and buttons for your Boolean controls and indicators, all accessible from the Modern>>Boolean and Classic>> Classic Boolean subpalette of the Controls palette (see Figures 4.25 and 4.26). You can change the state of a Boolean by clicking on it with the Operating tool. Like numeric controls and indicators, each type of Boolean has a default type based on its probable use (i.e., switches appear as controls, and LEDs as indicators).

Figure 4.25. Modern Boolean controls

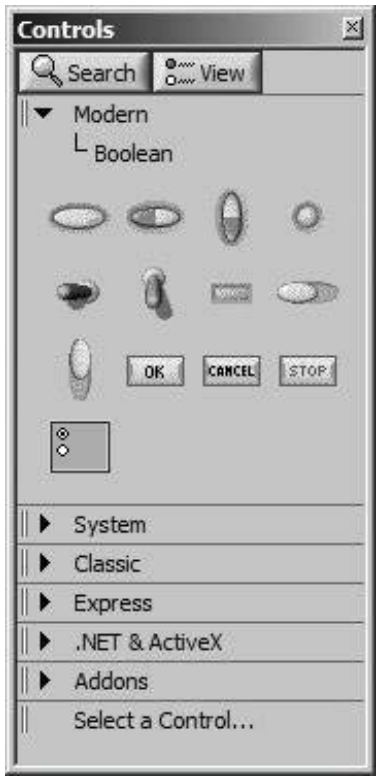
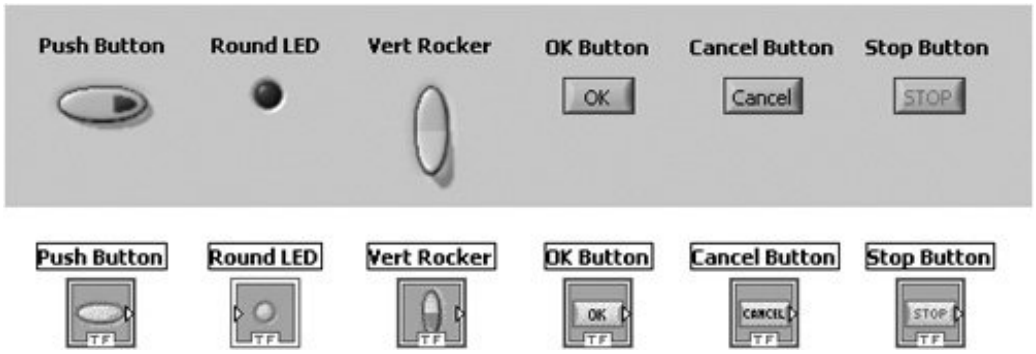


Figure 4.26. Classic Boolean controls

Boolean terminals appear green on the block diagram and contain the letters "TF." With the View As Icon option selected (from a terminal's pop-up menu), the terminal will be displayed as an icon that reflects the specific type of Boolean control or indicator, as shown in Figure 4.27.

Figure 4.27. Boolean controls and indicators (top) with terminals (bottom) in View As Icons mode

[View full size image]



Control terminals have thick borders along with an arrow pointing out from the right, while indicator borders are thin and have an arrow pointing in from the left. It is very important to distinguish between the two since they are not functionally equivalent (Control=Input=data source; Indicator = Output = data sink, so they are not interchangeable).

Labeled Buttons

LabVIEW has three buttons with text messages built into them: the OK, Cancel, and Stop buttons.

Not just these three, but all Booleans also have a Visible Items>>Boolean Text option that will display the word "ON," "OFF," depending on their states. This text is merely informative for the user. Each labeled button can contain two text messages: one for the TRUE state and one for the FALSE state. When you first drop buttons, the TRUE state says "ON" and the FALSE state says "OFF." You can then use the Labeling tool to change each message.

Note that clicking the mouse on the Boolean text of a button is the same as clicking on the button itself. However, the same is not true for a button's label or caption. If you move a label or caption over the button, clicking on the label or caption will have no resultit "blocks" the button from the user. This can be not only annoying, but dangerous, if a user cannot press an important button, such as one that aborts a machine operation.

Mechanical Action

A Boolean control has a handy pop-up option called Mechanical Action, which lets you determine how the Boolean behaves when you click on it (e.g., whether the value switches when you press the mouse button, switches when you release it, or changes just long enough for one value to be read and then returns to its original state). Mechanical Action is covered in more detail in Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs."

Customizing Your Boolean with Imported Pictures

You can design your own Boolean style by importing pictures for the TRUE and FALSE state of any of the Boolean controls or indicators. You can learn more about how to do this in Chapter 17, "The Art of LabVIEW Programming."

Strings

Simply put, string controls and indicators display text data. Strings most often contain data in ASCII format, the standard way to store alphanumeric characters. String terminals and wires carrying string data appear pink on the diagram. The terminals contain the letters "abc." You can find strings in the Modern>>String & Path and Classic>>Classic String & Path subpalettes of the Controls palette, shown in Figure 4.28 and Figure 4.29.

Figure 4.28. Modern string controls

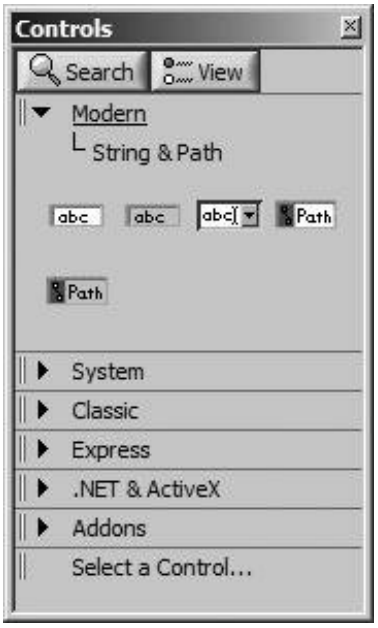
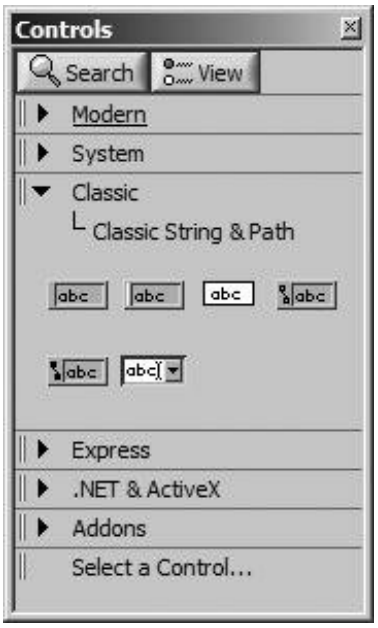


Figure 4.29. Classic string controls



Although string controls and indicators can contain numeric characters, they do NOT contain numeric data. You cannot do any numerical processing on string data; that is, you can no more add an ASCII "9" character than you can an "A." If you need to use numeric information that is stored in string format (to perform arithmetic, for example), you must first convert it to numeric format using the appropriate functions (see Chapter 9). It is a good practice to use only numeric controls and indicators for data that will contain numbers.

String controls and indicators are fairly simple. Their pop-up menus contain few special options:

- Enable Wrapping occurs where lines of text that are wider than the string control are "wrapped" around and continue on the next line of the string control. This option is enabled by default.
- Visible Items>>Vertical Scrollbar shows and hides the vertical scrollbar.
- Visible Items>>Horizontal Scrollbar shows and hides the horizontal scrollbar. This option is disabled when Enable Wrapping is enabled. (There is no reason to scroll horizontally, because

lines of text never extend past the right edge of the string control when wrapping is enabled.)

- Limit to Single Line prevents the user from adding multiple lines of text to a string control. Instead, pressing the <enter> or <return> key causes the value to be applied to the string control.
- Update Value While Typing causes the control value to change (as read from the terminal on the block diagram, etc.) each time the user types a keystroke. Otherwise, the user must apply the change by clicking outside the string control, pressing the apply button, etc.

We'll talk more about strings and their more complex cousin, the table, in Chapter 9.

Combo Box Control

The combo box control is similar to a text ring control, in that it has a list of strings; however, the value of the combo box is the selected string itself rather than the numeric index of the selected string within the list of strings. You can edit the list of strings, just as you would a ring controlpop up on the combo box and select Edit Items . . . from the shortcut menu. One very cool and little known feature of the combo box is that the user can type values directly into the text area and the combo box will auto-complete the text to match the first, shortest string from the items list that begins with the letters you type. And, you can select the Allow Undefined Strings option from the pop-up menu to allow users to enter values that are not in the list of items. This is a very powerful control that many people don't know about, so give it a try and impress your LabVIEW friends!

Paths

You use *path* controls and indicators to display paths to files, folders, or directories. If a function that is supposed to return a path fails, it will return <Not A Path> in the path indicator. Paths are a separate, platform-independent data type especially for file paths, and their terminals and wires appear bluish-green on the block diagram. A path is specified by drivename followed by directory or folder names and then finally the filename itself. On a computer running Windows, directory and file names are separated by a backslash (\) (see Figure 4.30); on Mac OS X, folder and file names are separated by a colon (:) (see Figure 4.31); on Linux machines, a forward slash (/) separates files and directories (see Figure 4.32). You'll learn more about paths in Chapter 9 and Chapter 14.

Figure 4.30. Windows path

Figure 4.31. Mac path

Figure 4.32. Unix path

Decorations

Just for fun, you can use LabVIEW's special Decorations subpalette of the Controls palette to enhance your front panel's appearance. These decorations have a solely aesthetic functionthey are the only objects from the Controls palette that do not have corresponding block diagram terminals (aside from the subpanel, which we will discuss in Chapter 13, "Advanced LabVIEW Structures and Functions").

Custom Controls and Indicators

To make programming even more fun, LabVIEW lets you create your own custom controls and indicators. So if LabVIEW doesn't provide exactly the one you want, make your own! You'll learn how to do that in Chapter 17.

Summary of Basic Controls and Indicators

Just to make sure you get your data types straight, we'll recap the four types of simple controls and indicators:

Numerics contain standard numeric values.

Booleans can have one of two states: on or off (true or false, one or zero).

Strings contain text data. Although they can contain numeric characters (such as zero to nine), you must convert string data to numeric data before you can perform any arithmetic on it.

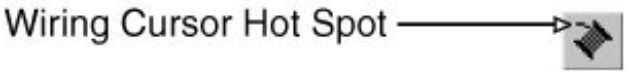
Wiring Up

Your neatly arranged front panel full of sharp-looking controls and indicators won't do you much good if you don't connect the wires in your diagram to create some action in your program. The following sections detail everything you need to know about wiring techniques.

Remember, wiring is always done on the block diagram, not the front panel.

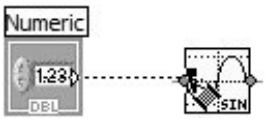
Wiring Tool

You use the Wiring tool to connect terminals. The cursor point or "hot spot" of the tool is the tip of the unwound wire segment, as shown.



To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and then click on the second terminal. It does not matter which terminal you click on first. The terminal area blinks when the hot spot of the Wiring tool is correctly positioned on the terminal, as shown in Figure 4.33. Clicking connects a wire to that terminal.

Figure 4.33. Wiring a numeric control's terminal to the input terminal of the Sine function



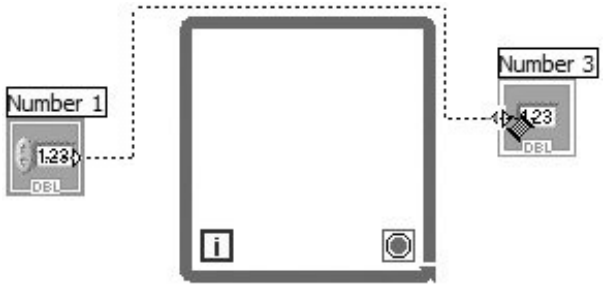
Once you have made the first connection, LabVIEW draws a wire as you move the cursor across the diagram, as if the wire were reeling off the spool. You do not need to hold down the mouse button.

To wire from an existing wire, perform the operation described previously, starting or ending the operation on the existing wire. The wire blinks when the Wiring tool is correctly positioned to fasten a new wire to the existing wire.

Automatic Wire Routing

It can be a lot of work to weave your wires in between objects that lay between your wire source and destination. To make this task easier, the LabVIEW Automatic Wire Routing feature automatically finds the best route for wires as you wire them, optimally routing the wire to decrease the number of bends in the wire (see Figure 4.34). You can temporarily disable automatic wire routing by pressing the <A> key after you start a wire. Pressing the <A> key will re-enable automatic wire routing. Also, you can clean up existing wires by right-clicking on a wire and selecting Clean Up Wire from the shortcut menu to automatically route them.

Figure 4.34. A wire automatically routing itself around object, with automatic wire routing enabled

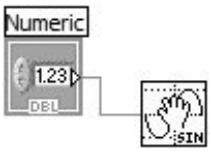


Some people find automatic wire routing annoying, not unlike Microsoft Word's auto-format features. If you want to disable automatic wire routing by default, you can do so under the LabVIEW options (uncheck the option at Tools>Options>>Block Diagram>>"Enable automatic wire routing").

Automatic Wiring

Another way you can wire functions is to use LabVIEW's automatic wiring feature. When you select a function from the Controls palette, you will notice that as you drag it over the block diagram, LabVIEW draws temporary wires to show you valid connections. If you drag the control near a terminal or other object that has a valid input or output, you'll notice LabVIEW connects the two (see Figure 4.35). Releasing the mouse button at this point "snaps" the wiring into place.

Figure 4.35. Automatic wiring



For auto-wiring to work, you will need to drag the function very, very close to the other object you are trying to wire it to. If the automatic wiring seems awkward or doesn't work for you, don't worry about itjust stick to "manual" wiring.

You can wire directly from a terminal outside a structure to a terminal within the structure using the basic wiring operation (you'll learn more about structures in Chapter 6, "Controlling Program Execution with Structures"). LabVIEW creates a tunnel where the wire crosses the structure boundary, as shown in Figures 4.36 and 4.37. Figure 4.36 shows what the tunnel looks like as you are drawing the wire; Figure 4.37 depicts a finished tunnel.

Figure 4.36. A wire as it is drawn from an object outside a structure to an object inside a structure, before the wire is connected

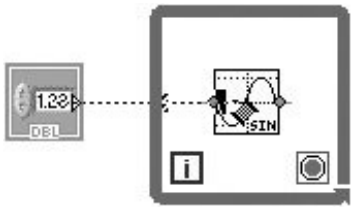
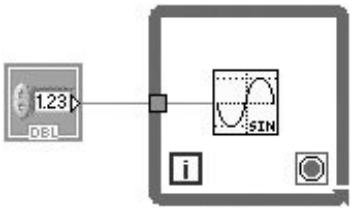


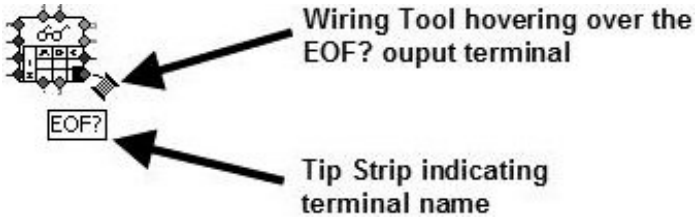
Figure 4.37. A wire that was drawn from an object outside a structure to an object inside a structure, which has a tunnel where the wire passes through the structure wall



Wiring Complicated Objects

When you are wiring a complicated built-in node or subVI, it helps to pay attention to the wire "whiskers" and tip strips that appear as the Wiring tool approaches the icon. Wire whiskers, the truncated wires shown around the VI icon in Figure 4.38, indicate the data type needed at that terminal by their style, thickness, and color. Whiskers can be used for the automatic wiring feature we just described.

Figure 4.38. Wire "wiskers" and tip strip that appear as the Wiring tool approaches the icon



You may also want to take advantage of the Context Help window feature that highlights each connector pane terminal. When you pass the Wiring tool over a terminal, the corresponding Context Help window terminal will blink so that you can be sure you are wiring to the right spot. You can also use the Context Help window to determine which connections are recommended, required, or optional.

Bad Wires

When you make a wiring mistake, a broken wirea black dotted line with a red "X" in the middle having arrows on either side indicating the direction of the data flowappears instead of the usual colored wire pattern. Until all such "bad wires" have been vanquished, your run button will appear broken and the VI won't compile.

Whether or not a red X is shown on broken wires is a configurable option. To change this setting, open the Tools>>Options menu, select Block Diagram from the top pull-down menu, and change the value of the Show red Xs on broken wires checkbox.

You can remove a bad wire by selecting and deleting it. A better method is to obliterate all bad wires at once by selecting Remove Broken Wires from the Edit menu or by using the keyboard shortcut, <control-B> under Windows and <command-B> on the Mac.

Broken wires can sometimes contain a lot of useful informationthey can be broken because of type conflicts that should be fixed, not because the wires need to be removed. Be very careful that you actually want to remove all the broken wires before using the Remove Broken Wires feature. And remember, if you make a mistake, you can use Undo (<ctrl-Z> in Windows, <meta-Z> in Linux, and <command-Z> in Mac OS X) to get the broken wires back.

Sometimes bad wires are mere fragments, hidden under something or so small you can't even see them. In some cases, all you need to do to fix a broken run arrow is Remove Broken Wires.

If you don't know why a wire is broken, hover your mouse over the broken wire and a text box will appear describing your problem(s). This text will also be displayed in the Context Help window. You can also click on the broken run button or pop up on the broken wire and choose List Errors. A dialog box will appear describing your problem(s). You can edit broken wires by popping up on the broken wire and choosing Delete Wire Branch, Create Wire Branch, Remove Loose Ends, Clean Up Wire, Change to Control, Change to Indicator, Enable Indexing at Source, and Disable Indexing at Source to correct the broken wire. Which of these options are available depends on the reason the wire is broken.

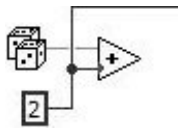
This procedure is for removing bad wires. If you have very bad and totally evil wires, then you must reboot your computer (just kidding!).

Wiring Tips

The following tips may make wiring a little easier for you:

- You can make a 90-degree turn in your wire, or "elbow" it, only once without clicking.
- Click the mouse to tack the wire and change direction.
- Change the direction from which the wire leaves a tack point by pressing the space bar.
- Double-click with the wiring tool to begin or terminate a wire in an open area.
- When wires cross, a small gap appears in the first wire drawn, as if it were underneath the second wire (see Figure 4.39). You can also choose Tools>>Options . . . and go to the Block Diagram menu; then check the box to Show dots at wire junctions.

Figure 4.39. Wires crossing each other, showing small white gaps in the bottom wire where the top wire crosses it



- Right mouse click to delete a wire while you're wiring, or <command>-click on the Mac.
- Use the Context Help window for more information about an object and to help you wire to the right terminal.

Create two numeric controls on a front panel and wire them to the inputs of the Add function. Don't wire the output just yet.

Wire Stretching

You can move wired objects individually or in groups by dragging the selected objects to the new location using the Positioning tool. Wires connected to the selected objects stretch automatically. If you duplicate the selected objects or move them from one diagram or subdiagram into another (for example, from the block diagram into a structure subdiagram such as a While Loop), LabVIEW leaves behind the connecting wires, unless you select them as well.

Wire stretching occasionally creates wire stubs or loose ends. You must remove these manually or by using the Remove Broken Wires command from the Edit menu (or the keyboard shortcut) before the VI will execute.

Now move the Add function with the Positioning tool and watch how the attached wires adjust.

Selecting and Deleting Wires

Positioning Tool

A wire segment is a single horizontal or vertical piece of wire. The point at which three or four wire segments join is a junction. A bend in a wire is where two segments join. A wire branch contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions in between. One mouse click with the Positioning tool on a wire selects a segment. A double-click selects a branch. A triple-click selects an entire wire. Press the <delete> or <backspace> key to remove the selected portion of wire.

Select and delete one of your wires; then rewire it.

Moving Wires

Positioning Tool

You can reposition one or more segments by selecting and dragging them with the Positioning tool. For fine tuning, you can also move selected segments one pixel at a time by pressing the arrow keys on the keyboard. LabVIEW stretches adjacent, unselected segments to accommodate the change. You can select and drag multiple wire segments, even discontinuous segments, simultaneously. When you move a tunnel, LabVIEW normally maintains a wire connection between the tunnel and the wired node.

Move a wire segment first using the Positioning tool, and then the arrow keys.

Wiring to Off-Screen Areas

If a block diagram is too large to fit on the screen, you can use the scroll bars to move to an off-screen area and drag whatever objects you need to that area. Dragging the Wiring tool slightly past the edge of the diagram window while you are wiring automatically scrolls the diagram (pressing shift will accelerate the scrolling). You can also click in empty space with the Positioning tool and drag outside the block diagram, and more space will be created.

Adding Constants, Controls, and Indicators Automatically

Instead of creating a constant, control, or indicator by selecting it from a palette and then wiring it manually to a terminal, you can pop up on the terminal and choose Create>>Constant, Create>>Control, or Create>>Indicator to automatically create an object with an appropriate

data type for that terminal. The new object will be automatically wired for you, assuming that makes sense. Remember this feature as you develop your programs, because it's amazingly convenient!

Create an indicator to display the results of your Add by popping up on the function and selecting Create>>Indicator. LabVIEW will create an indicator terminal wired to the Add output on the block diagram as well as a corresponding front panel indicator, saving you the effort.

PREV NEXT

Running Your VI

Run Button

You can run a VI using the Run command from the Operate menu, the associated keyboard shortcut, or by clicking on the Run button. While the VI is executing, the Run button changes appearance.

Run Button (Active)

The VI is currently running at its top level if the Run button is black and looks like it's "moving."

Run Button (subVI)

The VI is executing as a subVI, called by another VI, if the Run button has a tiny arrow inside the larger arrow.

Continuous Run Button

If you want to run a VI continuously, press the Continuous Run button, but be carefulthis is not a good programming habit to get into. You can accidentally catch your program in an endless loop and have to reboot to get out. If you do get stuck, try hitting the keyboard shortcut for the Abort command: <control-.> under Windows, <command-.> on Macs, and <alt-.> under Linux.

Abort Button

Press the Abort button to abort execution of the top-level VI. If a VI is used by more than one running top-level VI, its Abort button is grayed out. Using the Abort button causes an immediate halt of execution and is not good programming practice, as your data may be invalid. You should code a "soft halt" into your programs that gracefully wraps up execution. You will learn how very soon.

Pause Button

The Pause button pauses execution when you press it, and then resumes execution when you press it again.

You can run multiple VIs at the same time. After you start the first one, switch to the panel or diagram window of the next one and start it as previously described. Notice that if you run a subVI as a top-level VI, all VIs that call it as a subVI are broken until the subVI completes. You cannot run a subVI as a top-level VI and as a subVI at the same time.

Activity 4-2: Building a Thermometer

Now you're going to put together a VI that actually does something! You will build a simple program that simulates reading temperature in degrees Celsius, converts the value to degrees Fahrenheit, and displays both values.

Make sure you save this activity, because you will be adding to it later. If you didn't save it, you can find our version of `Thermometer.vi` *in* `EVERYONE\CH4` *when you need it*.

1. Open a new front panel.


Operating Tool

2. Drop a thermometer on the panel by selecting it from the Modern>>Numeric palette of the Controls menu. Label it `Temperature___(degF)` by typing inside the owned label box as soon as the thermometer appears on the panel.


Labeling Tool

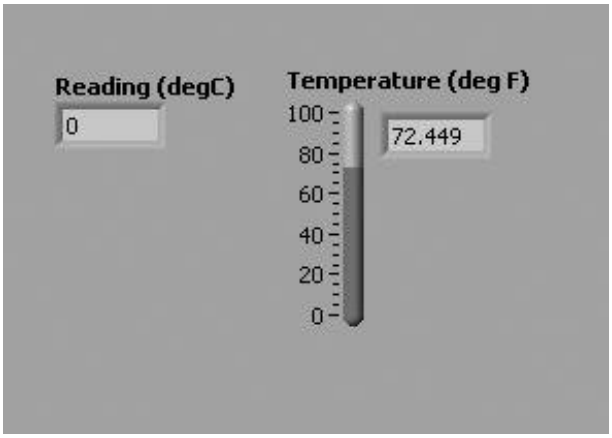
3. Make the exact value of the thermometer appear by right-clicking on the thermometer and choosing Visible Items>>Digital Display.


Positioning Tool

4. Now drop a Numeric Indicator from Modern>>Numeric Palette onto the panel. Label this indicator `Reading (degC).`

5. Remember, you can move controls and indicators with the Positioning tool to arrange them where you wish. Your front panel should look like the one in Figure 4.40.

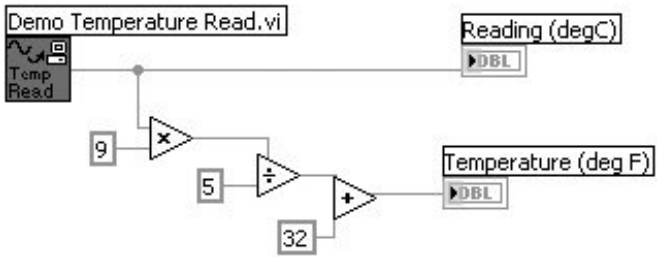
Figure 4.40. Thermometer.vi front panel



6. Save your VI as Thermometer.vi.

7. Build the block diagram shown in Figure 4.41. You might find it helpful to select Tile Left and Right from the Windows menu so that you can see both the front panel and the block diagram at the same time. We've provided you with a subVI in EVERYONE\CH04 called "Demo Read Temperature.vi." To put this subVI in your block diagram, choose Select a VI. . . from the Functions palette. Once you wire this, your block diagram should look like the one shown in Figure 4.41.

Figure 4.41. Block diagram of Thermometer.vi

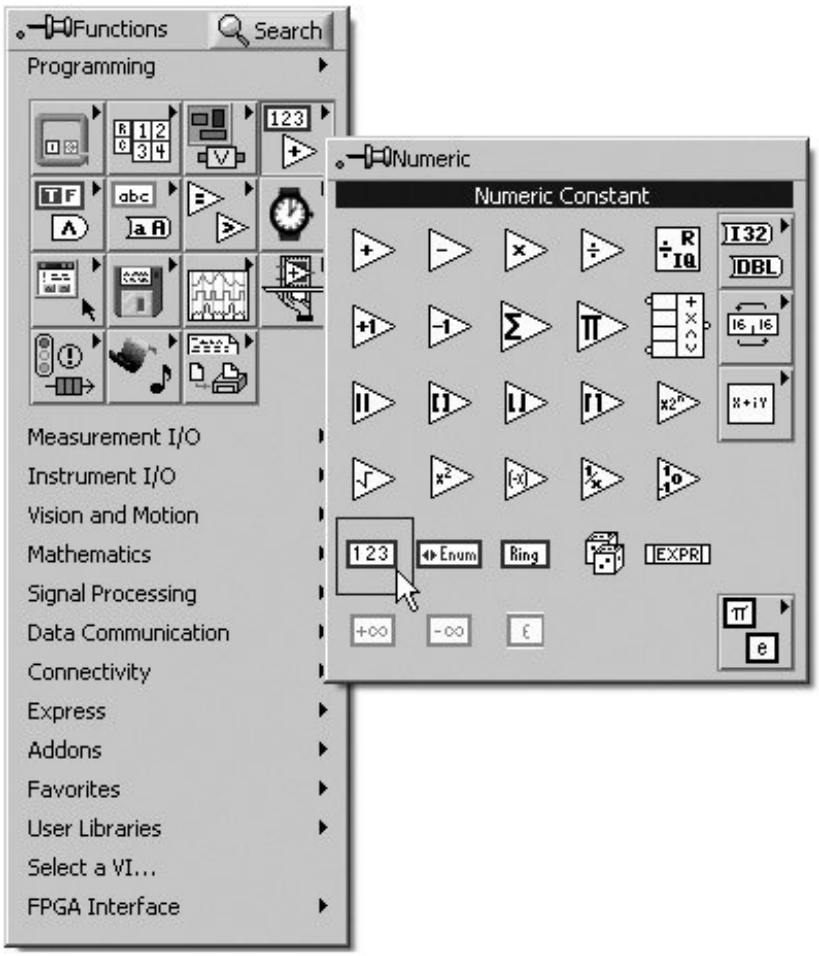


Remember, the subVI "Demo Temperature Read.vi" is in the EVERYONE\CH04 folder of the CD. This VI returns a temperature reading, different every time, in degrees Celsius. To convert to Fahrenheit, the formula is

$$\deg F = \frac{\deg C \cdot 9}{5} + 32$$

To implement this formula, use the Multiply, Divide, and Add functions found on the Programming>>Numeric palette. To create the block diagram numeric constants (9, 5, 32), use the Numeric Constant, also found on the Programming>>Numeric palette, as shown in Figure 4.42. (You can also create constants on the block diagram by selecting Create>>Constant from the pop-up menu of a function's terminal.)

Figure 4.42. Numeric constant on the Numeric palette



8. Run the VI several times by clicking on the Run button. You will see the thermometer display the temperature brought in from the simulation function. If you can't get your VI to compile, read Chapter 5, "Yet More Foundations," which explains debugging techniques. Then try again.


Run Button

9. Save the VI in your MYWORK directory by selecting Save from the File menu. Name it Thermometer.vi. You will be using this VI as a subVI later on in the book.

PREV NEXT

Useful Tips

As you do more and more programming in LabVIEW, you'll find some of these shortcuts very useful for putting together your VI more quickly. Look over all of these, and refer back to them later as reminders; you're sure to find some that will make you say, "I wish I'd known that!"

Keyboard Shortcuts

Many LabVIEW menu options have keyboard shortcuts. For example, to create a new front panel window, you can select the New VI option from the File menu or press the keyboard equivalent: <control-N> (for Windows), <meta-N> (for Linux), or <command-N> (for Mac OS X).

In general, the keyboard auxiliary keys, <ctrl> on Windows or <command> on Mac OS X, have the equivalent of <alt> key for Linux.

LabVIEW allows you to edit the menu shortcuts, modifying existing shortcut key combinations and assigning shortcut key combinations to those that do not have them by default. You can edit the menu shortcuts from the Tools>>Options menu in the Menu Shortcuts section.

Examples

Glance through the examples that ship in LabVIEW using the NI Example Finder, as discussed in Chapter 1, "What in the World Is LabVIEW?" You can use these programs as is or modify them to suit your application. You can open the NI Example Finder by selecting Find Examples . . . from the Help menu.

Changing Tools

When LabVIEW is in edit mode (and the *Lock Automatic Tool Selection* LabVIEW preferences option is not enabled, as described in Chapter 3, "The LabVIEW Environment"), pressing <tab> toggles through the tools. If the front panel is active, LabVIEW rotates from the Operating tool to the Positioning tool to the Labeling tool to the Color tool. If the block diagram is active, LabVIEW toggles through the tools in the same order, except that it selects the Wiring tool instead of the Color tool.

You can also press the spacebar to alternate between the Operating and Positioning tools in the front panel, and the Wiring and Positioning tools in the block diagram.

Changing the Direction of a Wire

Pressing the spacebar while you wire changes the direction from which the current wire branch leaves the last tack point. Thus, if you accidentally move horizontally from a tack point but want the wire to move down initially, pressing the spacebar changes the initial orientation from horizontal to vertical.

Canceling a Wiring Operation

To delete a wire as you are wiring, simply press the <Esc> key. Alternately, under Windows and Linux, click the right mouse button. On Mac OS X, wire off the screen and click.

Removing the Last Tack Point

Clicking while wiring tacks a wire down. <Shift>-clicking while wiring removes the last tack point, and <Shift>-clicking again removes the next-to-the-last tack point. If the last tack point is the terminal, <Shift>-clicking removes the wire.

Inserting an Object into Existing Wires

You can insert an object, such as an arithmetic or logic function, into an existing wiring scheme without breaking the wire and rewiring the objects. Pop up on the wire where you wish to insert the object, and choose Insert>>; then go ahead and choose the object you want to insert from the Functions palette that appears. If the inserted function has two input terminals of the same type (e.g., the subtract function), you can choose which gets wired by popping up slightly above (for the subtrahend) or below (for the minuend) the original wire.

Moving an Object Precisely

You can move selected objects very small distances by pressing the arrow keys on the keyboard once for each pixel you want the objects to move. Hold down the arrow keys to repeat the action. To move

the object in larger increments, hold down the <shift> key while pressing the arrow key.

Incrementing Digital Controls More Quickly

If you press the <shift> key while clicking on the increment or decrement button of a digital control, the display will increment or decrement very quickly. The size of the increment increases by successively higher orders of magnitude; for example, by ones, then by tens, then by hundreds, and so on. As the range limit approaches, the increment decreases by orders of magnitude, slowing down to normal as the value reaches the limit.

Entering Items in a Ring Control

To add items quickly to ring controls, press <shift-enter> or <shift-return> after typing the item name to accept the item and position the cursor to add the next item.

Cloning an Object

To clone objects, select the objects to be copied, hold down the <control> key (Windows), <meta> key (Linux), or the <option> key (Mac OS X), and drag the duplicates to the new position. The original objects remain where they are. You can clone objects into another VI window as well.

Moving an Object in Only One Direction

If you hold down the <shift> key while moving or cloning objects, LabVIEW restricts the direction of movement horizontally or vertically, depending on which direction you move the mouse first.

Matching the Color

To pick a color from an object, click on the object with the Color Copy tool. Then color other objects by clicking on them using the Color tool.

Replacing Objects

You can easily replace a front panel or block diagram object by popping up on it and selecting Replace>>. When replacing functions on the block diagram, a Functions palette will appear beneath the All Palettes>> submenu, where you can choose a new function. The new one will replace the old, and any wires that are still legal will remain intact. The Replace>> menu will also have a submenu allowing to access the palette where the object being replaced is located. For example, if you are replacing an array function, the Replace>>Array Palette submenu will be present. When replacing front panel objects, there are no submenus, just the entire Controls palette from which to make your choice.

Making Space

To add more working space to your panel or diagram window, hold down the <control> key in Windows, <option> key in Mac OS X, or <meta> key on Linux machines, and drag out a region with the Positioning tool. You will see a rectangle marked by a dotted line, which defines your new space. When you are happy with the size and location of the rectangle, release the mouse button to create the new space. If you wish to cancel the operation while it is in progress, press the <escape> key.

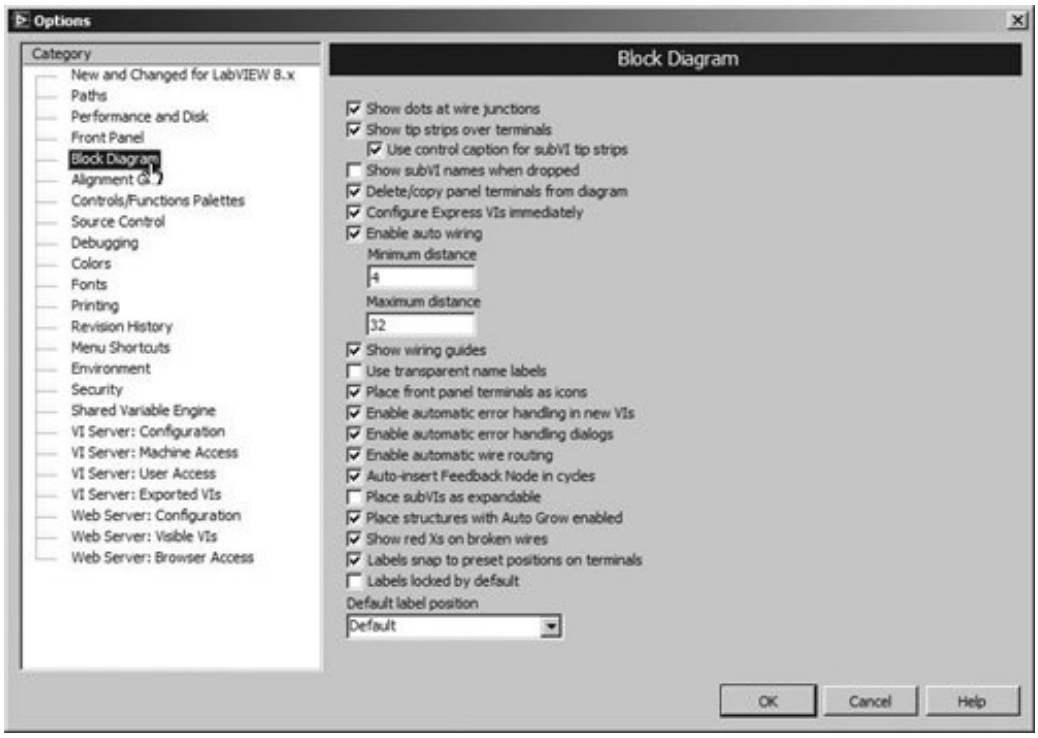
You can use this feature to easily create space between objects on your block diagram and front panel, but don't overuse ityou can quickly create a sloppy-looking block diagram (with lots of wire bends, which is considered bad "style") by adding space in this way.

Configuring Your Preferences

LabVIEW has many preferences and options you can configure to suit your taste and convenience by selecting Options . . . from the Tools menu. You can select which options category to view by selecting it from the Category list on the left side of the Options dialog window (see Figure 4.43).

Figure 4.43. Options dialog

[View full size image]



Select Options . . . from the Tools menu and browse through the different options available to you. If you want to know more about options, look in the LabVIEW manuals or online help.

PREV NEXT

Wrap It Up!

LabVIEW has special editing tools and techniques fitting to its graphical environment. The Operating tool changes an object's value. The Positioning tool selects, deletes, and moves objects. The Wiring tool creates the wires that connect diagram objects. The Labeling tool creates and changes owned and free labels. Owned labels belong to a particular object and cannot be deleted or moved independently, while free labels have no such restrictions.

LabVIEW has four types of simple controls and indicators: *numeric*, *Boolean*, *string*, and *path*. Each holds a separate data type and has special pop-up options. Control and indicator terminals and wires on the block diagram are color coded according to data type: floating-point numbers are orange, integer numbers are blue, Booleans are green, strings are pink, and paths are bluish-green.

You place objects on the front panel or block diagram using the Controls or Functions palette, respectively. You can also access these palettes by popping up in an empty section of the panel or diagram.

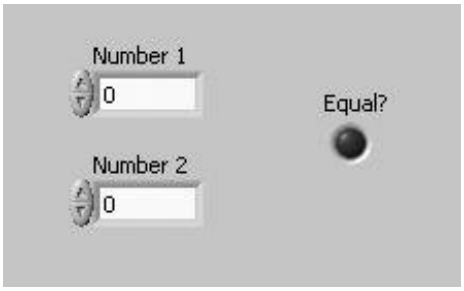
To run your VI, click on the run button or select Run from the Operate menu. If your run button is broken, it means something is wrong in your VI. Read the next chapter to learn good debugging techniques.

Additional Activities

Activity 4-3: Comparison Practice

Build a VI that compares two input numbers. If they are equal, an LED on the front panel turns on. Name it Comparison Practice.vi (see Figure 4.44).

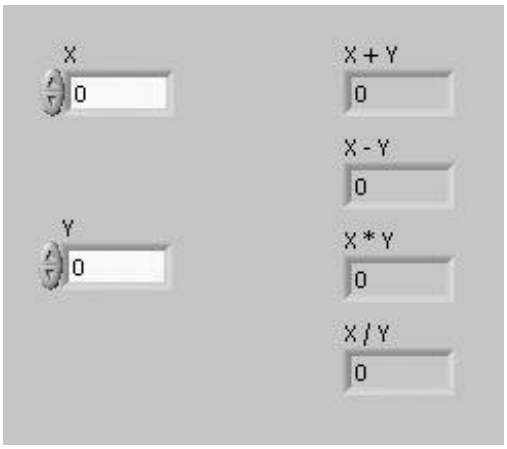
Figure 4.44. Comparison Practice.vi front panel



Activity 4-4: Very Simple Calculator

Build a VI that adds, subtracts, multiplies, and divides two input numbers and displays the result on the front panel. Use the front panel in Figure 4.45 to get started. Call it Very Simple Calculator.vi.

Figure 4.45. Very Simple Calculator.vi front panel



If you get stuck, you can find the answers in `EVERYONE\CH04.`

PREV NEXT

5. Yet More Foundations

Overview

In this chapter, you will learn more fundamentals of the LabVIEW environment. We'll talk about load and save basics, LabVIEW's special LLB files, debugging features, subVI implementation, and documentation procedures.

Goals

- Be able to load and save your VIs (and then save regularly!)
- Learn how to make LabVIEW's powerful debugging features work for you
- Learn how to use probes
- Create your very first subVI and understand how it's used
- Document your achievements for all the world to see

Key Terms

- LLB file
- Broken VI
- Single-step mode
- Node
- Execution highlighting
- Probe
- Breakpoint
- SubVI
- Tip strip
- Control description
- VI description
- Icon Editor
- Required, recommended, and optional inputs

Loading and Saving VIs

Obviously, you will be loading and saving VIs quite a bit during your development. LabVIEW has many features that can accommodate your file storage needs, and this section discusses how you can make them work for you.

You can load a VI by selecting Open from the File menu, and then choosing the VI from the dialog box that appears. As the VI loads, you may see a status window that lists the subVIs that are currently being loaded and allows you to cancel the loading process. You can load a specific VI by double-clicking on the VI's icon (either in the LabVIEW Project Explorer, or directly from the operating system's file manager) or by dragging the icon on top of the LabVIEW icon. If LabVIEW is not already running, it will first launch and then open the VI.

Save VIs by selecting Save (or a similar option) from the File menu, or use the keyboard shortcut <ctrl-S> (Windows), <command-S> (Mac OS X), or <meta-S> (Linux). LabVIEW then pops up a file dialog box so you can choose where you want to save

Keep in mind that LabVIEW references VIs by name. *You cannot have two VIs with the same name in memory at one time (unless they are members of different project libraries, which we will discuss in Appendix D, "LabVIEW Object-Oriented Programming"*). When searching for a VI of a given name, LabVIEW will load the first VI it finds, which may not be the one you intended.

Note that an asterisk (*) marks the titles of VIs that you've modified but not yet saved (see Figure 5.1). We're sure you already know to save your work constantly, but we'd like to stress the importance of saving frequently and backing up everything you do on a computeryou never know when lightning will strike (literally)!

Figure 5.1. A VI window title with an asterisk (*) indicating that the VI has unsaved changes

Never save your VIs in the `vi.lib` directory. This directory is updated by National Instruments during new version releases of LabVIEW, and if you put anything in there, you may lose your work.

Save Options

You can save VIs with one of four save options in the File menu.

Select the Save option to save a new VI and then specify a name for the VI and its destination in the disk hierarchy; or use this option to save changes to an existing VI in a previously specified location. Save All works just like Save except that it applies to all VIs in memory that have unsaved changes.

Save As . . . brings up the dialog box, shown in Figure 5.2, in which you can choose to Copy or Rename the VI.

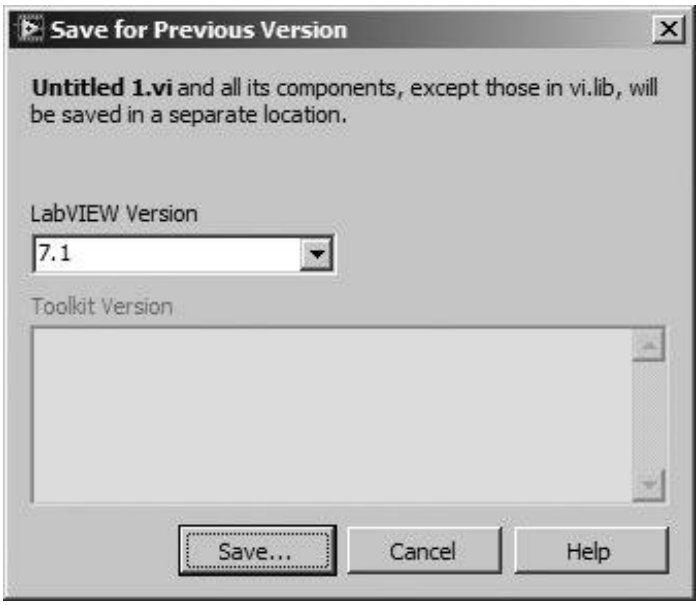
- Copy, which creates a copy on disk, has three sub-options:
    - Substitute copy for original This is a "traditional" Save As . . . operationit renames the VI in memory and saves a copy of the VI to disk under the new name. All VIs currently in memory that call the old VI now point to the new VI.
    - Create unopened disk copy This simply creates a copy on disk. It does not open the copy into memory.
    - Open additional copy This creates a copy on disk, and then opens the copy. Unlike Substitute copy for original, all VIs currently in memory that call the old VI still point to the old VI.
- Rename renames the VI in memory and then moves (renames) the file on disk. Note that the original file is deleted. All VIs currently in memory that call the VI now point to the VI in its new location.

Figure 5.2. Save As dialog

[View full size image]

Save for Previous Version . . . brings up a dialog box, shown in Figure 5.3, in which you can choose to save the VI and all of its components to a previous version of LabVIEW.

Figure 5.3. Save for Previous Version dialog



Revert

You can use the Revert . . . option in the File menu to return to the last saved version of the VI you

are working on. A dialog box will appear to confirm whether you want to discard any changes in the VI.

LLBs

LLBs are special LabVIEW files that have the same load, save, and open capabilities as directories and folders within the LabVIEW environment. You can group several VIs together and save them as an LLB, much like you would store files inside one zip file. LLBs offer few advantages and many disadvantages; for example, they can contain only LabVIEW file types, not data or other files. In addition, your operating system sees *LLB files* as single files, and you can access their contents only from LabVIEW.

Note that many of the VIs shipped with LabVIEW are kept inside LLB files.

In the last few years, the trend has been to no longer use LLBs. LLBs were created in the early days of Windows in part to handle the old limitations of eight-character filenames, and have remained to this day for compatibility purposes. Storing your VIs as individual files, organized in directories, will usually make more sense.

LLB files were at one time referred to as "LabVIEW Libraries," hence the name and file extension "`.llb`." However, LabVIEW now has the `project library`, which is a logical collection of VIs used in LabVIEW projects. In comparison, LLB files are simply aggregates of LabVIEW files on disk. In this book, when we use the term "library," we are referring to `project libraries` and will exclusively use the term "LLB" for referring to LLB files.

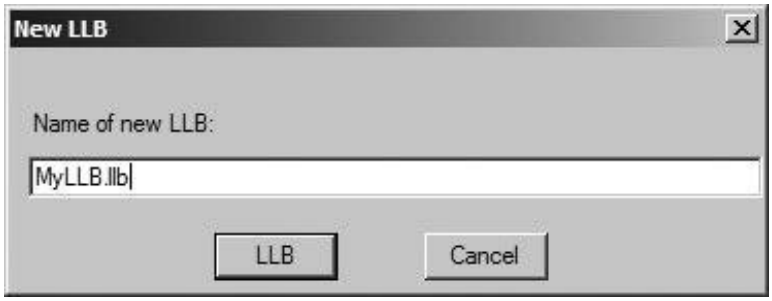
For the activities in this book, we've asked you to save your work in a `MYWORK` directory, so that you can access individual files more easily. We also encourage you to not use LLB files for storing your day-to-day work. The LabVIEW Project Explorer, which you learned about in Chapter 3, "The LabVIEW Environment," provides you with a great way to organize your VIs and other project files.

How to Use LLBs

Create an LLB from the Save or Save As . . . dialog box by clicking on the New LLB button under Windows or the New . . . button on Mac OS X. If you're on a Mac configured to use native dialog boxes, you will have to click on the Use LLBs button from the Save dialog box, and then select New . . . from the dialog box that appears.

Enter the name of the new LLB in the dialog box that appears, shown in Figure 5.4, and append an `.llb` extension. Then click on the LLB button, and the LLB file is created. If you do not include the `.llb` extension, LabVIEW adds it.

Figure 5.4. New LLB dialog



Usually you will create an LLB when you are saving a VI, so after the LLB file is created, a dialog box appears to let you name your VI and save it in your new LLB.

Once you've created an LLB, you can save VIs in it and access them through LabVIEW much like a directory or folder, but you cannot see the individual VIs from your operating system (unless you are on a Windows system and have checked the Enable Windows Explorer for LLB files option in the Environment section of the Tools>>Options . . . dialog). Remember that on Macs configured to use native dialogs, you'll have to select Use LLBs from the Save dialog box in order to access them.

On Windows, LabVIEW installs a File Explorer extension that allows you to navigate into LLB files as if they were folders on disk. This feature is enabled by default, and is controlled by the Enable Windows Explorer for LLB Files setting in the Environment category of the Tools>>Options . . . dialog. If you change this setting, you will need to reboot your computer before you will see the effect of the change.

The LLB Manager

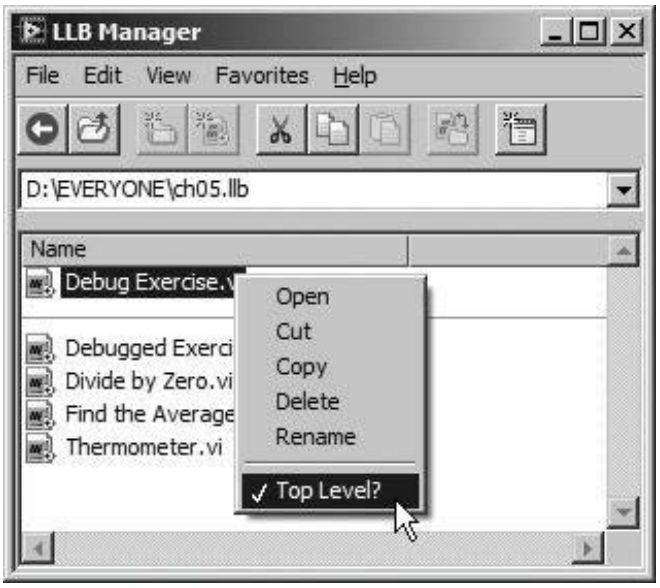
Because you can't do it through your operating system, you must use the LLB Manager to edit the contents of an LLB. You can use the LLB Manager (from the Tools menu) to simplify copying, renaming, and deleting files within LLBs as well as within your file system itself. You also can use this tool to create new LLBs and directories and convert LLBs to and from directories. Creating new LLBs and directories and converting LLBs to and from directories is important if you need to manage your VIs with source code control tools. (Yes, you can use the *Windows Explorer for LLB Files* feature, but it does not provide all the features of the LLB Manager.)

You can open more than one instance of the LLB Manager and drag and drop files between the instances to move or copy the files from one location to the other. Holding down the <Ctrl> key while you drop a file will cause the file to be copied rather than moved.

You can open VIs from the LLB Manager by double-clicking them.

You can also use the LLB Manager to modify the Top Level attribute of VIs within an LLB. If you mark a VI as Top Level, it will load automatically when you open the LLB file, as shown in Figure 5.5. You can have more than one top-level VI in an LLB file. Top-level VI names will also appear in a separate section at the top of the Load dialog, so it's easier for you to determine which VIs are main VIs and which are subVIs.

Figure 5.5. LLB Manager



Save and Load Dialogs

LabVIEW supports the File dialog box format that your system uses. When you open or save a VI, your system File dialog box appears. If you click on an LLB file (a special LabVIEW file storage structure that can contain VIs (as well as some other LabVIEW file types) within it, similar to how a ZIP file can contain other files), LabVIEW replaces the system dialog box with its own File dialog box so that you can select files within the LLB.

Because the interface for selecting and saving into an LLB is somewhat awkward with the system dialog box, you may prefer to set the LabVIEW options to use the standard LabVIEW dialog box if you commonly use LLB files. Select Options . . . from the Tools menu, then go to the Environment menu and uncheck the Use native file dialogs box.

The standard LabVIEW dialog box also saves shortcuts to recently selected folders, which are available from a pull-down menuthis can save you time. However, using the LabVIEW Project Explorer to manage your project VIs makes this feature less important.

The Save dialog box in Mac OS X disables files from LLBs. You will need to click the Use LLBs button in order to save into a LabVIEW LLB file, or use the standard LabVIEW dialog box instead of the native one.

Filter Rings

At the bottom of your Save or Load dialog box, you will see a filter ring that allows you to view All Files, view only All LabVIEW Files, VIs & Controls, VIs, Controls, Templates, Projects, Run Time Menu Files, Project Libraries, or XNodes files, or just see those with a Custom Pattern that you specify.

The Custom Pattern option is not available on some native-style dialog boxes.

If you choose Custom Pattern, another box appears in which you can specify a pattern. Only files matching that pattern will show up in the dialog box. Notice that an asterisk automatically appears inside the box; this is the "wild-card character" and is considered a match with any character or characters.

PREV NEXT

Debugging Techniques

Have you *ever* written a program with no errors in it? LabVIEW has many built-in debugging features to help you develop your VIs. This section explains how to use these conveniences to your best advantage.

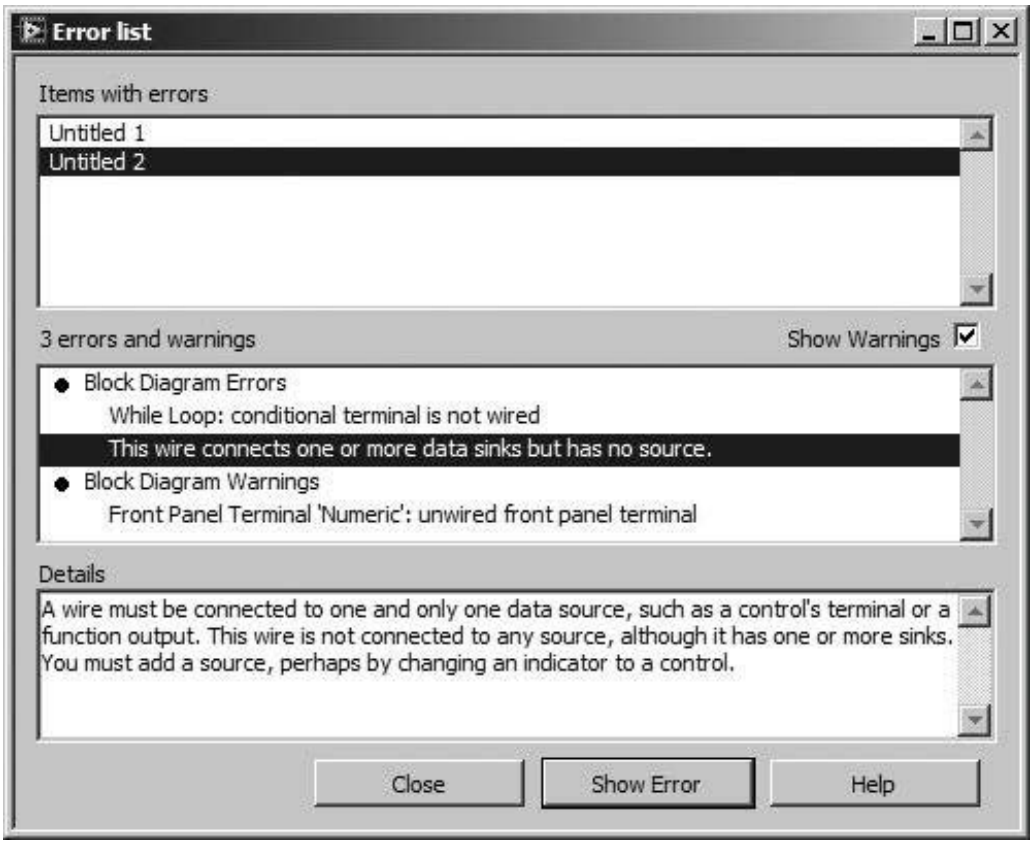
Fixing a Broken VI

Run Button (broken)

A *broken VI* is a VI that cannot compile or run. The Run button appears as a broken arrow to indicate that the VI has a problem. It's perfectly normal for a VI to be broken while you are creating or editing it, until you finish wiring all the icons in the diagram. Sometimes you may need to Remove Broken Wires (found in the Edit menu) to clean up loose wires, but be careful not to delete wires you want!

To find out why a VI is broken, click on the broken Run button or select Show Error List from the Windows menu. An information box titled "Error List" appears, listing all errors for the VI. You can choose to see the error list for other open VIs using a menu ring at the top of the window. To find out more about a particular error, click on it. The Error List window will display more information. To locate a particular error in your VI, double-click on the error in the list or highlight it and press the Show Error button. LabVIEW brings the relevant window to the front and highlights the object causing the error (see Figure 5.6).

Figure 5.6. Error List dialog



Warnings

Warning Button

If you want extra debugging help, you can choose to Show Warnings in the Error List window by clicking in the appropriate box. A warning is something that's not illegal and won't cause a broken run arrow, but does not make sense to LabVIEW, such as a control terminal that is not wired to anything. If you have Show Warnings checked and have any outstanding warnings, you will see the Warning button on the Toolbar. You can click on the Warning button to see the Error List window, which will describe the warning.

You can also configure LabVIEW's options to show warnings by default. Go to the Debugging section in the Options dialog box (accessed by selecting Tools>>Options . . .) and check the Show warnings in Error List by default box.

Most Common Mistakes

Certain mistakes are made more frequently than others, so we thought we'd list them to make your life easier. If your Run button is broken, one of these might describe your problem. Please see Appendix E, "Resources for LabVIEW" for a link to the *LabVIEW FAQ*, a great on-line resource containing answers to frequently asked questions.

- A function terminal requiring an input is unwired. You cannot leave unwired functions on the diagram while you run a VI to try out different algorithms.
- The block diagram contains a bad wire due to a data-type mismatch or a loose, unconnected end, which may be hidden under something or is so tiny that you can't see it. The Remove Broken Wires command from the Edit menu eliminates the bad wires, but you might have to look a little harder to find a data-type conflict.
- A subVI is broken, or you edited its connector after placing its icon on the diagram. Use the Replace or Relink to subVI pop-up option to re-link to the subVI.
- You have a problem with an object that is disabled, invisible, or altered using a property node (which we'll talk more about in Chapter 13).
- You have unwittingly wired two controls together, or wired two controls to the same indicator. The Error List window will bear the message, "You have connected a Control to a Control. Change one to an indicator," or "This wire connects more than one data source," for this problem. You can often solve it by changing one of those controls to an indicator.
- A subVI cannot be found, either because you don't have it (maybe someone sent you a VI without including the subVI) or because you changed the name of the subVI on disk and the calling VIs are not aware of the name change. When you change a VI's name using the Rename option of the File>>Save As . . . menu option, any caller in memory will "feel" the change and relink to the renamed VI. You must save each caller of the renamed subVI in order for the caller to "remember" the new name and location of the renamed subVI, the next time the caller is loaded from disk.

Single-Stepping Through a VI

Pause Button

For debugging purposes, you may want to execute a block diagram node by node. *Nodes* include subVI's, functions, structures, code interface nodes (CINs), formula nodes, and property nodes. To begin single-stepping, you can start a VI by clicking on one of the single-step buttons (instead of the Run button), pause a VI by setting a breakpoint, or click on the Pause button. To resume normal execution, hit the Pause button again.

You may want to use execution highlighting (described next) as you single-step through your VI, so you can visually follow data as it flows through the nodes.

While in *single-step mode*, press any of the three step buttons that are active to proceed to the next

step. The step button you press determines how the next step will be executed.

Step Into Button

Press the *Step Into* button to execute the first step of a subVI or structure and then pause at the next step of the subVI or structure. Or, you can use the keyboard shortcut: down arrow key in conjunction with <control> under Windows and <command> on Macs and <meta> on Linux.

Step Over Button

Press the *Step Over* button to execute a structure (sequence, loop, etc.) or a subVI and then pause at the next node. Or, you can use the keyboard shortcut: right arrow key in conjunction with <control> under Windows and <command> on Macs.

Step Out Button

Press the *Step Out* button to finish executing the current block diagram, structure, or VI and then pause. Or, you can use the keyboard shortcut: up arrow key in conjunction with <control> under Windows and <command> on Macs.

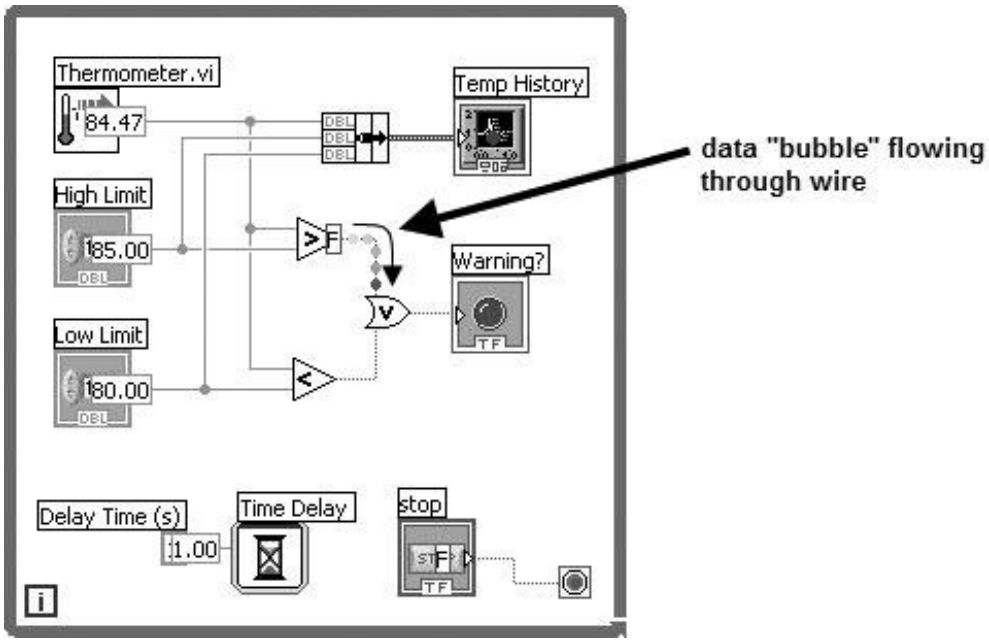
Execution Highlighting

Execution Highlighting Button

Sometimes it's nice to see exactly where your data is and what's happening to it. In LabVIEW, you can view an animation of VI block diagram execution. To enable this mode, click on the *Execution Highlighting* button in the Toolbar.

As data passes from one node to another, the movement of data is marked by bubbles moving along the wires. You will notice that highlighting greatly reduces the performance of a VI. Click again on the Execution Highlighting button to resume normal execution. Figure 5.7 shows a VI running with execution highlighting enabled.

Figure 5.7. The block diagram of a running VI that has Execution Highlighting enabled, showing "data bubbles" flowing through wires and the values of wires appearing in tip strips



Node values are automatically shown during execution highlighting, as in the previous illustration, if you select Auto probe during execution highlighting from the Debugging menu of the Options dialog.

You commonly use execution highlighting in conjunction with single-step mode to gain an understanding of how data flows through nodes. When these two *modes* are used together, execution glyphs on the subVI's icon indicate which VIs are running and which are waiting to run.

Be aware that data in independent (parallel) "chunks" of the diagram (those not having data flow dependencies) may not always flow in the same sequence on subsequent executions or iterations. And sometimes "race conditions" that cause problems under normal execution (with execution highlighting turned off) may disappear completely with execution highlighting turned on and appear to behave as "expected," thus hiding the buggy behavior. A race condition refers to a situation in your program where two or more independent processes are trying to write data at the same time to a variable (thus, they "race" against each other). You will see more examples of race conditions, and how to avoid them, in the latter chapters of this book. If you see such quirky behavior, look for race conditions as the possible cause.

Setting Breakpoints

Don't panic*breakpoints* do not "break" a VI; they only suspend its execution (similar to the *Pause* button) so that you can debug it. Breakpoints are handy if you want to inspect the inputs to a VI, node, or wire during execution. When the diagram reaches a breakpoint, it activates the pause button; you can single-step through execution, probe wires to see their data, change values of front panel objects, or simply continue running by pressing the Pause button or the Run button.

Breakpoint Tool

To set a breakpoint, click on a block diagram object with the Breakpoint tool from the Tools palette. Click again on the object to clear the breakpoint. The appearance of the breakpoint cursor indicates whether a breakpoint will be set or cleared.

Depending on where they are placed, breakpoints behave differently:

- If the breakpoint is set on a *block diagram*, a red border appears around the diagram and the pause will occur when the block diagram completes.

  Set Breakpoint Cursor

- If the breakpoint is set on a *node*, a red border frames the node and execution pauses just before the node executes.
- If the breakpoint is set on a *wire*, a red bullet appears on the wire and any attached probe will be surrounded by a red border. The pause will occur after data has passed through the wire.

  Clear Breakpoint Cursor

When a VI pauses because of a breakpoint, the block diagram comes to the front, and the object causing the break is highlighted with a marquee.

Breakpoints are saved with a VI but only become active during execution.

Saving VIs with breakpoints is generally not advised, because they are intended for debugging your VIs. It is easy to inadvertently save a VI after setting a breakpoint. If you

accidentally do so, you may be puzzled the next time you or a colleague opens your LabVIEW project and finds that a saved breakpoint is suspending execution of a VI.

You can easily find breakpoints in your VIs using the Find dialog, which can be opened by selecting Edit>>Find and Replace from the menu or using the shortcut <ctrl-F> (Windows), <command-F> (Mac OS X), or <meta-F> (Linux). From the Find dialog, select Search for: Objects, Object: Others>>Breakpoint, and define your Application Instance and Search Scope options.

Suspending Execution

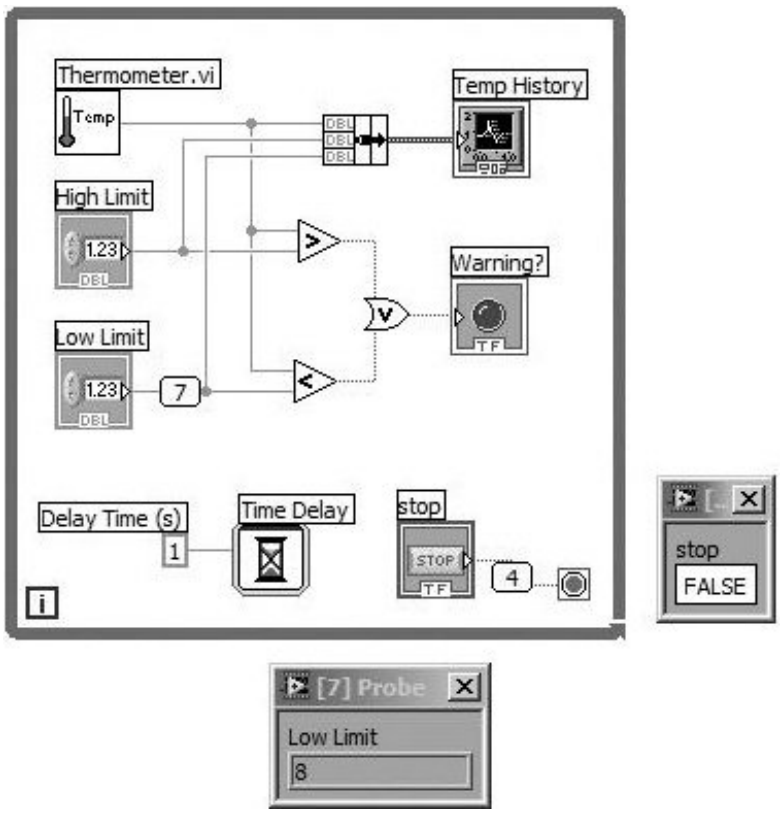
You can also enable and disable breakpoints with the Suspend when Called option, found in the Execution menu of VI Properties . . . (which you access from the icon pane pop-up menu in a VI's front panel). Suspend when Called causes the breakpoint to occur at all calls to the VI on which it's set. If a subVI is called from two locations in a block diagram, the subVI breakpoint suspends execution at each call.

If you want a breakpoint to suspend execution only at a *particular* call to the subVI, set the breakpoint using the SubVI Node Setup . . . option. Pop up on the subVI icon (in the block diagram of the calling VI) to access this option. You will learn more about VI setup options in Chapter 15, "Advanced LabVIEW Features."

Using the Probe

Use the *probe* to check intermediate values in a VI that executes but produces questionable or unexpected results. For instance, assume you have a diagram with a series of operations, any one of which may be the cause of incorrect output data. To fix it, you could create an indicator to display the intermediate results on a wire, or you can leave the VI running and simply use a probe. To access the probe, select the Probe tool from the Tools palette and click its cursor on a wire, or pop up on the wire and select Probe. The probe display, which is a floating window, first appears empty if your VI is not running (unless you have pressed the *Retain Wire Values* button, which we discuss shortly). When you run the VI, the probe display shows the value carried by its associated wire. Figure 5.8 shows a VI block diagram with probes placed on two of the wires.

Figure 5.8. A block diagram with probes placed on two of the wires



You can use the probe with execution highlighting and single-step mode to view values more easily. Each probe and the wire it references are automatically numbered uniquely by LabVIEW to help you keep track of them. A probe's number will not be visible if the name of the object probed is longer than the Probe window itself; if you lose track of which probe goes with which wire, you can pop up on a probe or a wire and select Find Wire or Find Probe, respectively, to highlight the corresponding object.

You cannot change data with the probe.

PREV NEXT

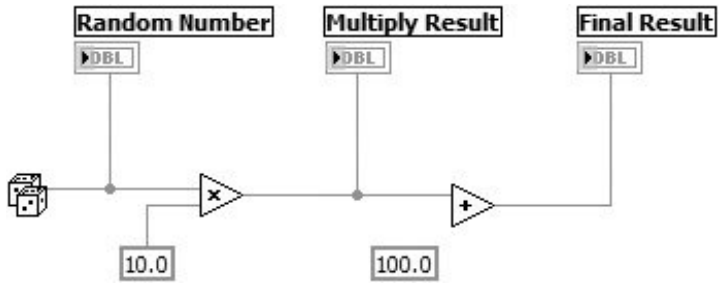
Activity 5-1: Debugging Challenge

In this activity, you will troubleshoot and fix a broken VI. Then you will practice using other debugging features, including execution highlighting, single-step mode, and the probe.

1. Open up the VI called Debug Exercise.vi, located in `EVERYONE\CH05`.

2. Switch to the block diagram, which is shown in Figure 5.9. Notice that the run arrow is broken. You must find out why and rectify the situation so the VI will run.

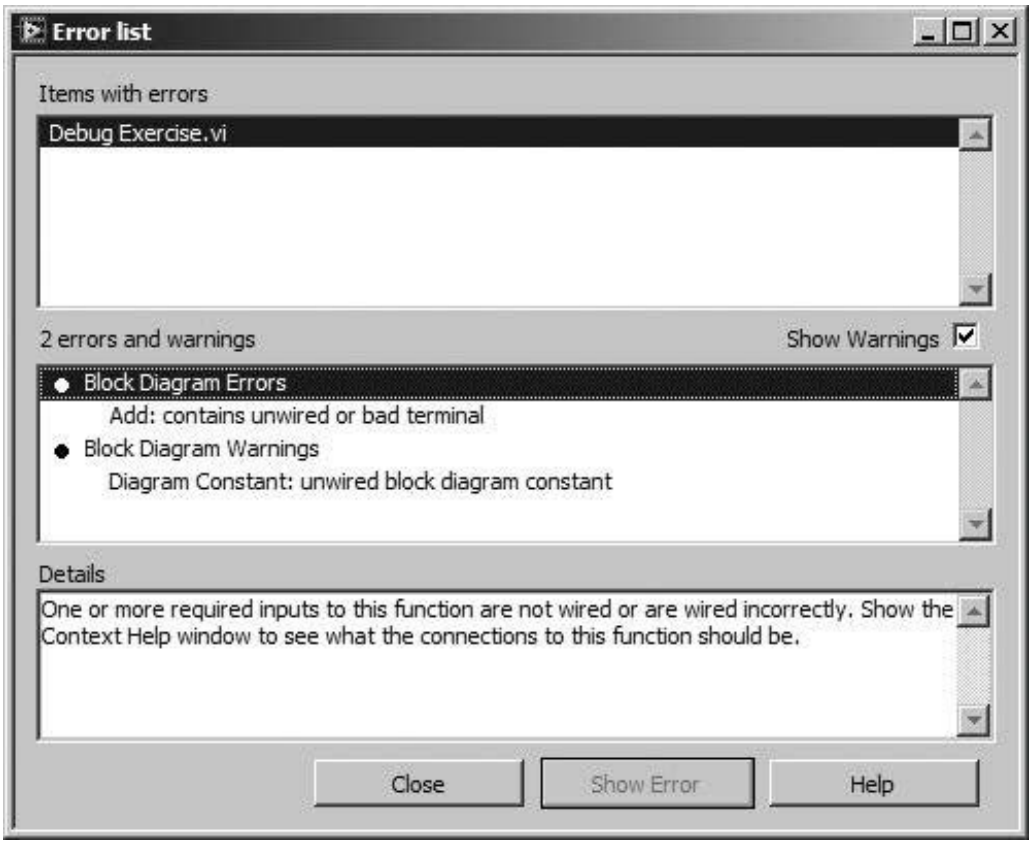
Figure 5.9. Debug Exercise.vi block diagram, which you must debug during this activity



3. Click on the broken run arrow. An Error List dialog box appears, describing the errors in the VI.

4. Click on the "Add: contains unwired or bad terminal" error. The Error List window, shown in Figure 5.10, will give you a more detailed description of the error. Now double-click on the error, or click the Find button. LabVIEW will highlight the offending function in the block diagram, to help you locate the mistake.

Figure 5.10. Error list dialog showing the errors that are present in Debug Exercise.vi and prevent it from running



5. Draw in the missing wire. The Run button should appear solid. If it doesn't, try to Remove Broken Wires.

6. Switch back to the front panel and run the VI a few times.

7. Tile the front panel and block diagram (using the Tile command under the Windows menu) so you can see both at the same time. Enable execution highlighting and run the VI in single-step mode by pressing the appropriate buttons on the Toolbar in the block diagram.


Execution Highlighting Button

8. Click the Step Over button each time you wish to execute a node (or click the Step Out button to finish the block diagram). Notice that the data appears on the front panel as you step through the program. First, the VI generates a random number and then multiplies it by 10.0. Finally, the VI adds 100.0 to the multiplication result. Notice how each of these front panel indicators is updated as new data reaches its block diagram terminals, a perfect example of dataflow programming. Remember, you can exit single-step mode and complete the VI by pressing the Pause button. Also notice that the tip strips describing the single-step buttons change text to give you an exact description of what they will do when you click them, given the context of where you are.

Step Into Button

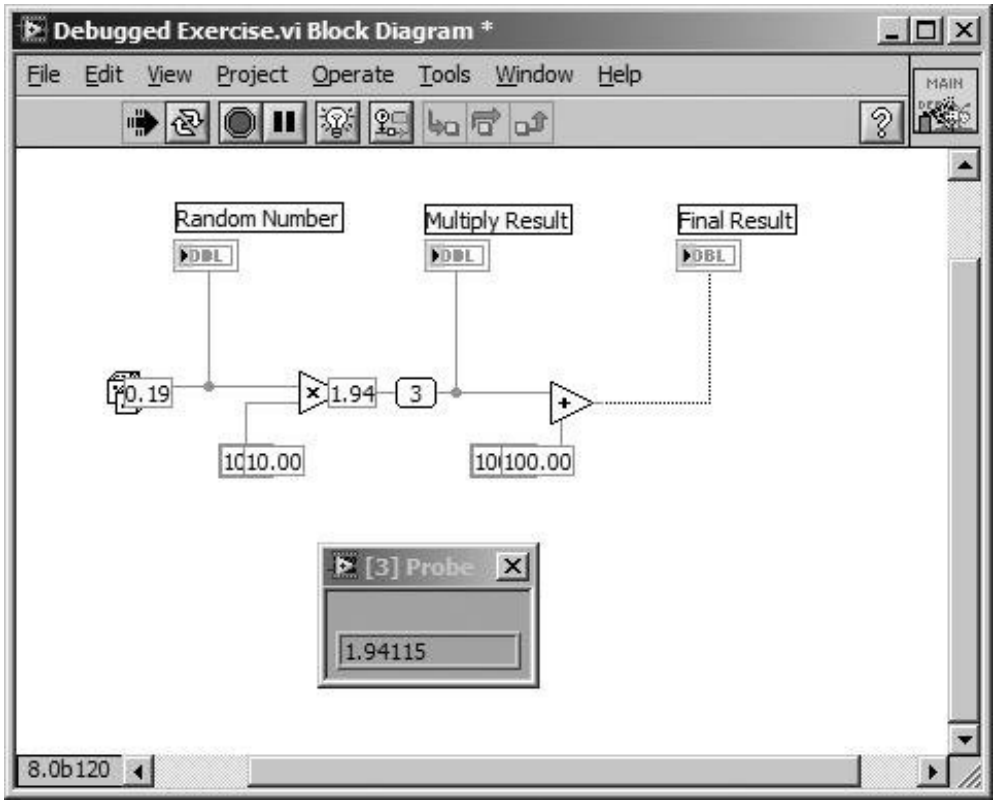
Step Over Button

Step Out Button

9. Now enable the probe by popping up on any wire segment and selecting Probe.

10. Step through the VI again and note how the probe displays the data carried by its corresponding wire, as shown in Figure 5.11.

Figure 5.11. Debugged Exercise.vi block diagram



11. Turn off execution highlighting by clicking its button. You're almost done . . . .

12. Save your finished VI in your MYWORK directory by selecting the Save As . . . option from the File menu so you won't overwrite the original. Name it Debugged Exercise.vi. If you're feeling adventuresome, or if you think an LLB will suit your needs, try creating an LLB and saving your work again in it for practice.

13. Close the VI by selecting Close from the File menu. Good job!

PREV NEXT

Creating SubVIs

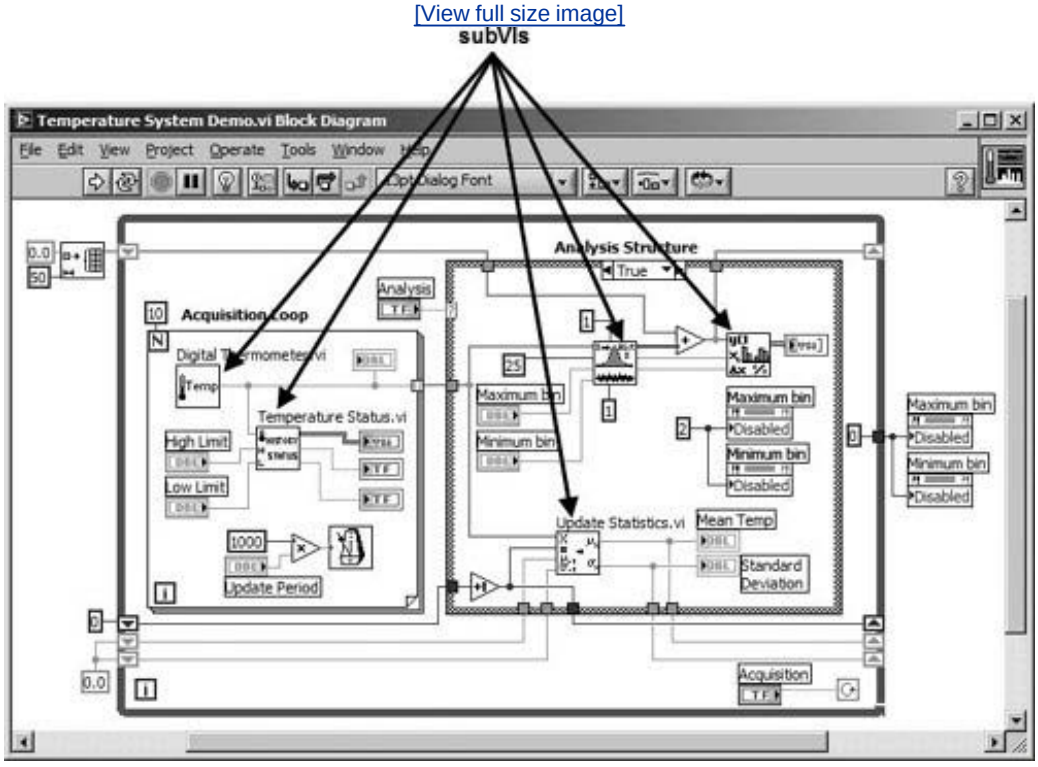
Much of LabVIEW's power and convenience stems from its modularity. You can build the parts of your program one complete module at a time by creating subVIs.

A subVI is simply a VI used in (or called by) another VI. A subVI node (comprised of icon/connector in a calling VI block diagram) is analogous to a subroutine call in a main program.

Figure 5.12 shows how the block diagram of Temperature System Demo.vi calls several subVIs.

Figure 5.12. SubVIs are like functions or subprograms called by VIs.

[View full size image]

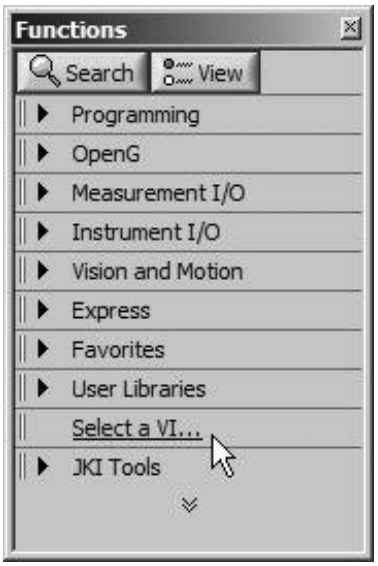


A block diagram can contain several identical subVI nodes that call the same subVI several times.

You can use any VI as a subVI in the block diagram of another VI, provided its icon has been created and its connector assigned. Drop existing VIs on a block diagram to use as subVIs with the Select a

VI . . . button in the Functions palette (see Figure 5.13). Choosing this option produces a file dialog box from which you can select any VI in the system; its icon will appear on your diagram.

Figure 5.13. The Select a VI . . . button on the Functions palette



A VI can't call itself directly by calling itself as a subVI. If you really need to do this, and implement recursion, you can have a VI call itself indirectly using a VI reference, which we'll talk about in Chapter 15.

Creating a SubVI from a VI

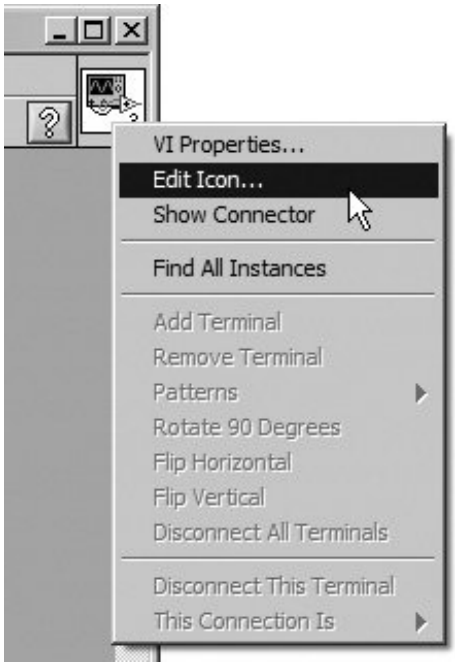
Before you use a VI as a subVI, you must supply a way for the VI to receive data from and pass data to the calling VI. To do this, you need to assign the VI controls and indicators to terminals on its connector pane, and you must create an icon to represent the VI.

Designing the Icon

Every subVI must have an icon to represent it in the block diagram of a calling VI; the icon is its graphical symbol. You can create the icon by selecting Edit Icon from the pop-up menu of the icon

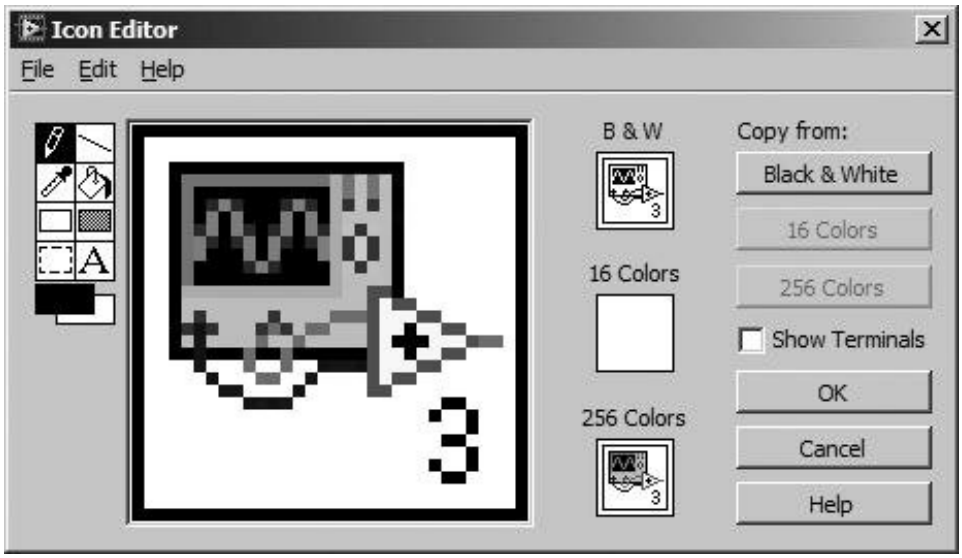
pane in the upper-right corner of the front panel (see Figure 5.14). You must be in edit mode to get this menu.

Figure 5.14. Selecting the Edit Icon . . . option from the icon pane of a VI, in order to open the Icon Editor dialog



You can also access the *Icon Editor* by double clicking on the icon in the icon pane. The Icon Editor window, shown in Figure 5.15, will appear. Use its tools to design the icon of your choice.

Figure 5.15. The Icon Editor dialog, which allows you to edit a VI's icon



	Pencil	Draws and erases pixel by pixel. Press <shift> to restrict drawing to horizontal, vertical, and diagonal lines.
	Line	Draws straight lines. Press <shift> to restrict drawing to horizontal, vertical, and diagonal lines.
	Color Copy (Dropper)	Copies the foreground color from an element in the icon. Use the <shift> key to select the background color with the dropper.
	Fill (Fill Bucket)	Fills an outlined area with the foreground color.
	Rectangle	Draws a rectangle in the foreground color. Double-click on this tool to frame the icon in the foreground color. Use the <shift> key to constrain the rectangle to a square shape.
	Filled Rectangle	Draws a rectangle bordered with the foreground color and filled with the background color. Double-click on this tool to frame the icon in the foreground color and fill it with the background color.
	Select	Selects an area of the icon for moving, cloning, or other changes. Double-click on this tool to *Select All* pixels.
	Text	Enters text into the icon design. Double-click on this tool to change the font attributes.
	Foreground/Background	Displays the current foreground and background colors. Click on each to get a palette from which you can choose new colors.

Press and hold down the <control> (Windows), <option> (Mac OS X), or <meta> (Linux) key to momentarily change any tool to the Color Copy tool (and release the key to change back). This trick allows you to quickly "grab" another foreground color without having to change tools.

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

- OK Saves your drawing as the VI icon and returns to the front panel window.
- Cancel Returns to the front panel window without saving any changes.
- Help Opens the LabVIEW Help documentation for the Icon Editor.

Although rarely used anymore, you can design a separate icon for display in monochrome, 16-color, and 256-color mode (this feature is an inheritance from the old days of LabVIEW where some people would need to run it on monochrome or 16-color monitors). You can design and save each icon version separately; you can also copy an icon from color to black and white (or vice versa) using the Copy from . . . buttons. Your VIs must always have at least a black and white icon. If no black and white icon exists, LabVIEW will force you to use a default icon.

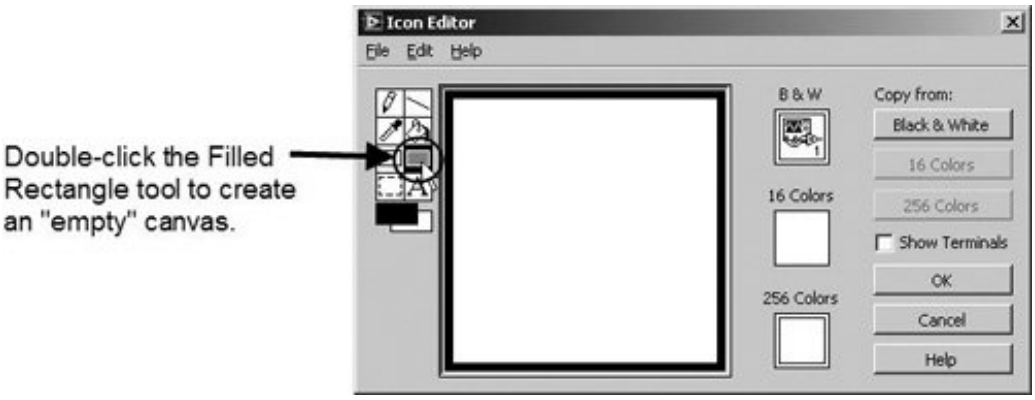
It is a common practice to create an "icon banner" (a distinctive portion of the icon across the top that has text and a background color) and use it across the top of all VIs that belong to a library, class, or module (any collection of VIs that all "belong" in a group). This icon banner will help you quickly identify that your VIs belong to that group.

To create an icon banner, first open a new VI and then open the Icon Editor.

1. Double-click the Filled Rectangle tool to draw a solid black border (foreground) around a solid white background, as shown in Figure 5.16.

Figure 5.16. Double-clicking the Filled Rectangle tool to create an "empty" canvas

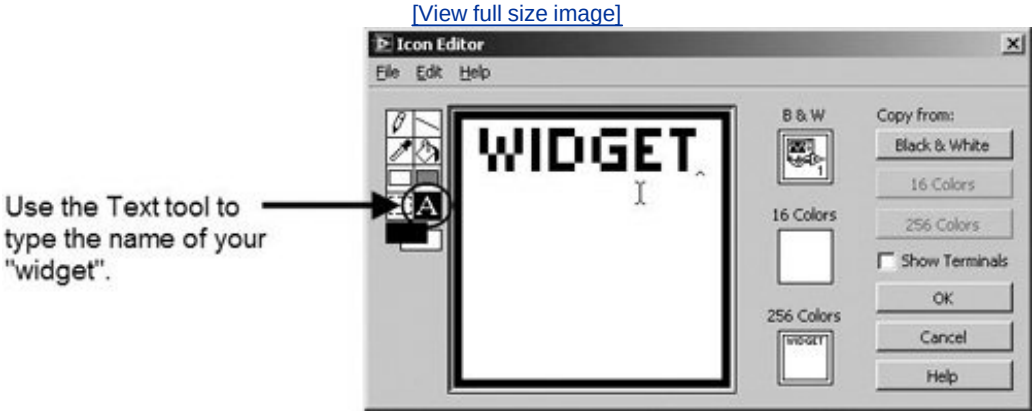
[View full size image]



2. Select the Text tool. If you are on Windows, we recommend you use the font named "Small Fonts" and 8-point font size. You can double-click the Text tool to open the Text Tool Font dialog to make this selection. Place the cursor anywhere in the icon canvas and type "WIDGET" (in all uppercase letters). After typing this text, and before you change tools or click the mouse anywhere, reposition the text by pressing the arrow keys (up, down, left, and right) to position the text, as shown in Figure 5.17. See how the text moves? (Note that it is usually easier to position the text with the arrow keys, after inserting it, than it is to perfectly place the text cursor before typing.)

Figure 5.17. Using the Text tool to type the "icon banner" text

[View full size image]



3. Select the Line tool and draw a horizontal line below the text, as shown in Figure 5.18.

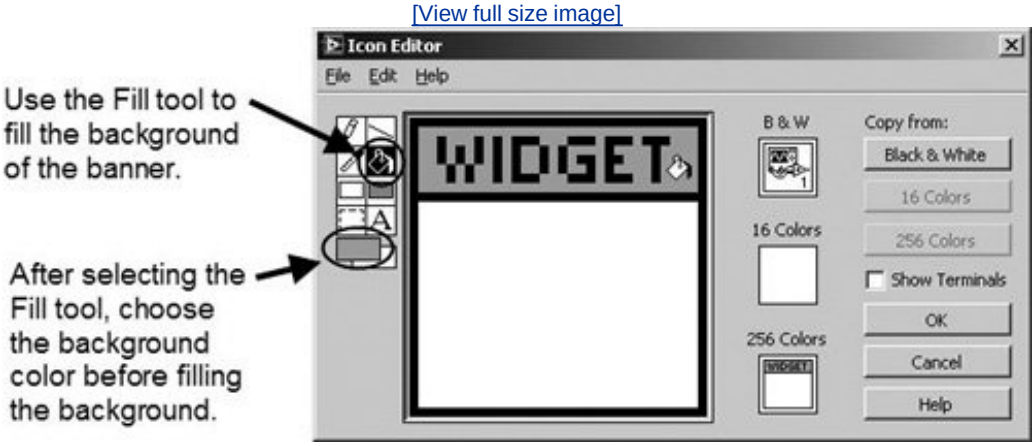
Figure 5.18. Using the Line tool to draw a horizontal line under the "icon banner" text

[View full size image]

4. Select the Fill tool and then choose a color for the background color of your banner. Make sure that the color contrasts with the color you selected for the banner text. Now fill the background color of the banner, as shown in Figure 5.19.

Figure 5.19. Using the Fill tool to fill the background of the "icon banner"

[View full size image]



Congratulationsyou now have an icon banner. If you want, you can use this VI as a template for creating other VIs in your "widget" component. Or, you can just copy the banner, using the Select tool, and paste it into other icons after opening the Icon Editor for the target VI.

You can choose different text and background colors for each library, class, module, etc., in your project.

For each VI in a group, place specific text or graphics in the open area beneath the icon banner that is unique to that VI and signifies the VI's function.

Assigning the Connector

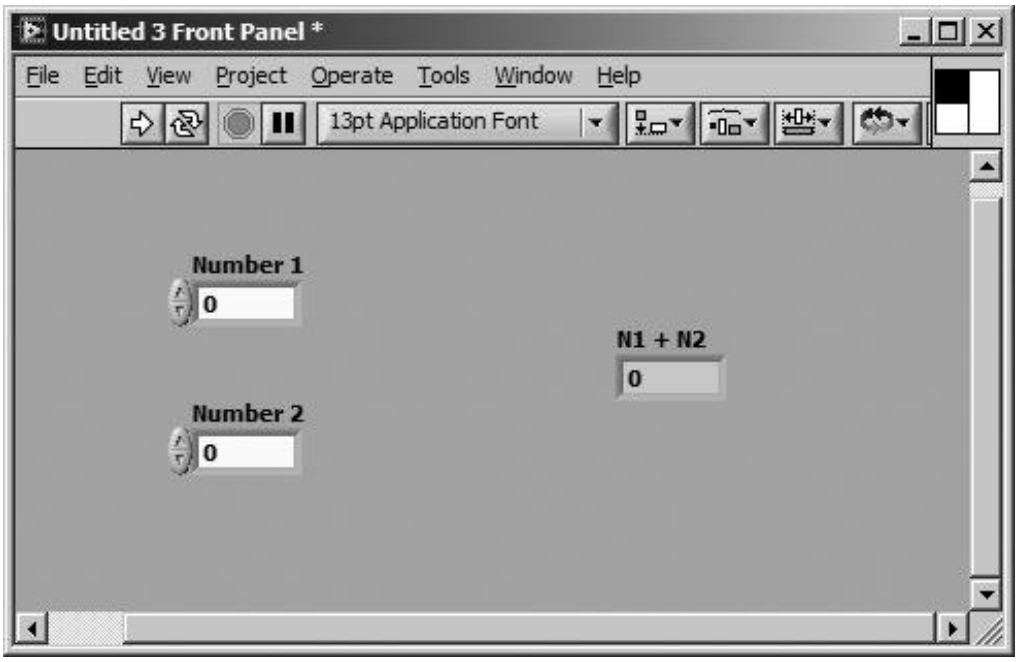
Before you can use a VI as a subVI, you will need to assign connector terminals. The connector is LabVIEW's way of passing data into and out of a subVI, just like you must define parameters for a subroutine in a conventional language. The connector of a VI assigns the VI's control and indicators to input and output terminals. To define your connector, pop up in the icon pane and select Show Connector (when you want to see the icon again, pop up on the connector and select Show Icon). LabVIEW chooses the default connector, having 12 terminals. If you want a different one, choose it from the Patterns menu, obtained by popping up on the connector. You can also rotate and flip your connector if it doesn't have a convenient orientation, using commands in the connector pop-up menu.

It is generally recommended (and considered good "style" by many experienced LabVIEW developers) to use the default connector pane (which has 12 terminals) and to use the left-side terminals for inputs (controls) and the right-side terminals for outputs (indicators). Having 12 terminals will give you "spare" (unused) terminals, which you can assign to controls or indicators at a later time.

Follow these steps to assign a terminal to a control or indicator:

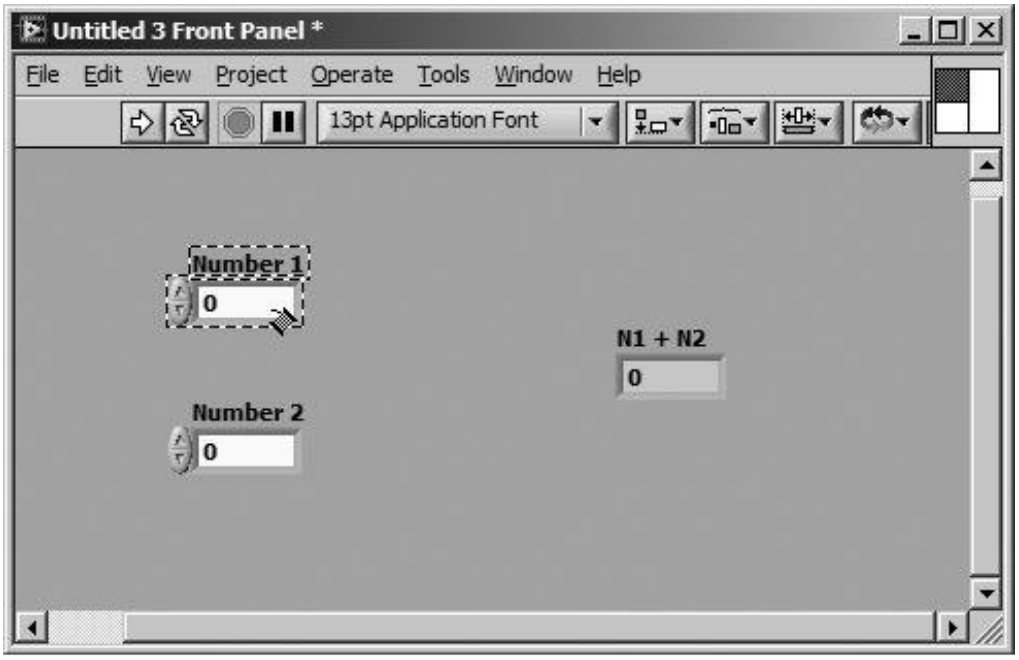
1. Click on a terminal in the connector. The cursor automatically changes to the Wiring tool, and the terminal turns black, as shown in Figure 5.20.

Figure 5.20. Your VI with the upper left connector pane terminal selected for wiring to a front panel control



2. Click on the control or indicator you want that terminal to represent. A moving dotted line frames the control or indicator (see Figure 5.21).

Figure 5.21. Your VI with the upper left connector pane terminal wired to the **Number 1** control



3. After you have successfully assigned the control or indicator to the connector pane terminal, it will appear orange, the color LabVIEW uses for floating point numbers (or the different color for the data type of the control or indicator that you've chosen).

If the terminal is white or black, you have not made the connection correctly. Repeat the previous steps if necessary. You can have up to 28 connector terminals for each VI.

You can reverse the order of the first two steps, but you must choose the wiring tool first.

If you make a mistake, you can Disconnect a particular terminal or Disconnect All by selecting the appropriate action from the connector's pop-up menu.

Creating SubVIs from a Block Diagram Selection

Sometimes you won't realize you should have used a subVI for a certain section of code until you've already built it into the main program. Fortunately, you can also create subVIs by converting a part of the code in an existing VI. Use the Positioning tool to select the section of the VI that you want to replace with a subVI, choose Create SubVI from the Edit menu, and watch LabVIEW replace that section with a subVI, complete with correct wiring and an icon. You can double click on your new subVI to view its front panel, edit its icon, look at its connector, and save it under its new name. Use Create SubVI with caution, as you may cause some unexpected results. You will learn more about this handy feature in Chapter 15, "Advanced LabVIEW Features," where we'll describe the restrictions that apply as well as common pitfalls.

SubVI Help: Recommended, Required, and Optional Inputs

If you bring up the Help window on a subVI node in a block diagram, its description and wiring pattern will appear. Input labels appear on the left, while outputs appear on the right. You can also bring up the Help window on the current VI's icon pane to see its parameters and description. You will learn how to edit the description in the next section.

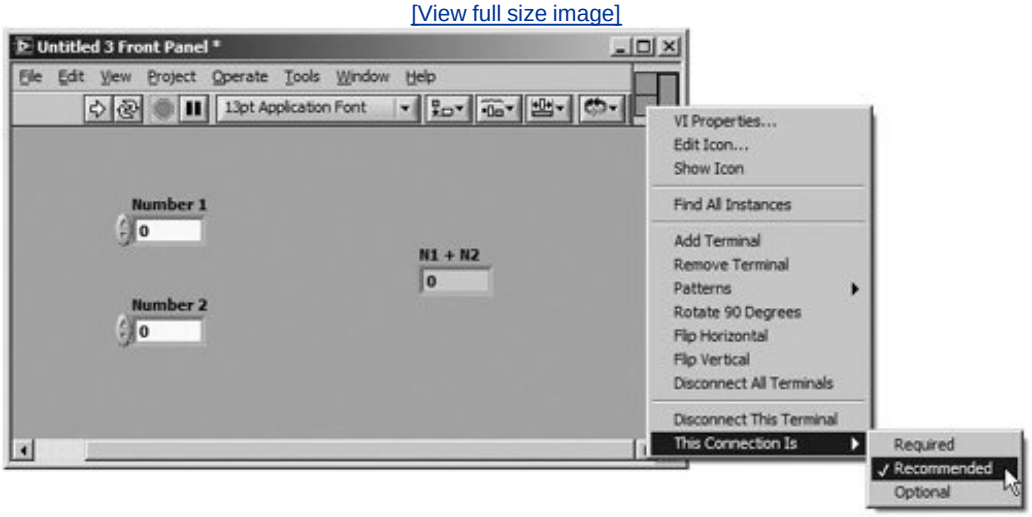
Built-in LabVIEW functions automatically detect if you have not wired a required input and break the VI until you do. You can also configure your subVIs to have the same types of required, recommended, and optional inputs as functions. When an input is *required*, you cannot run the VI as a subVI without wiring that input correctly. When an input or output is *recommended*, you can run

the VI, but the Error List window will list a warning (if you have warnings enabled) that tells you a recommended input or output is unwired. When an input is *optional*, no restrictions are enforced, and the connection is often considered advanced.

To mark a connection as required, recommended, or optional (or see what state it's in currently), pop up on the assigned terminal in the connector pane and take a look at the This Connection Is>> pullout menu. A checkmark next to Required, Recommended, or Optional indicates its current state (see Figure 5.22).

Figure 5.22. A connector pane terminal shown as Recommended, from the pop-up menu

[View full size image]



In the Help window, required connections appear bold, recommended connections are in plain text, and optional connections are grayed if you are using detailed help view. If the Help window is set to simple help view, optional connections are hidden.

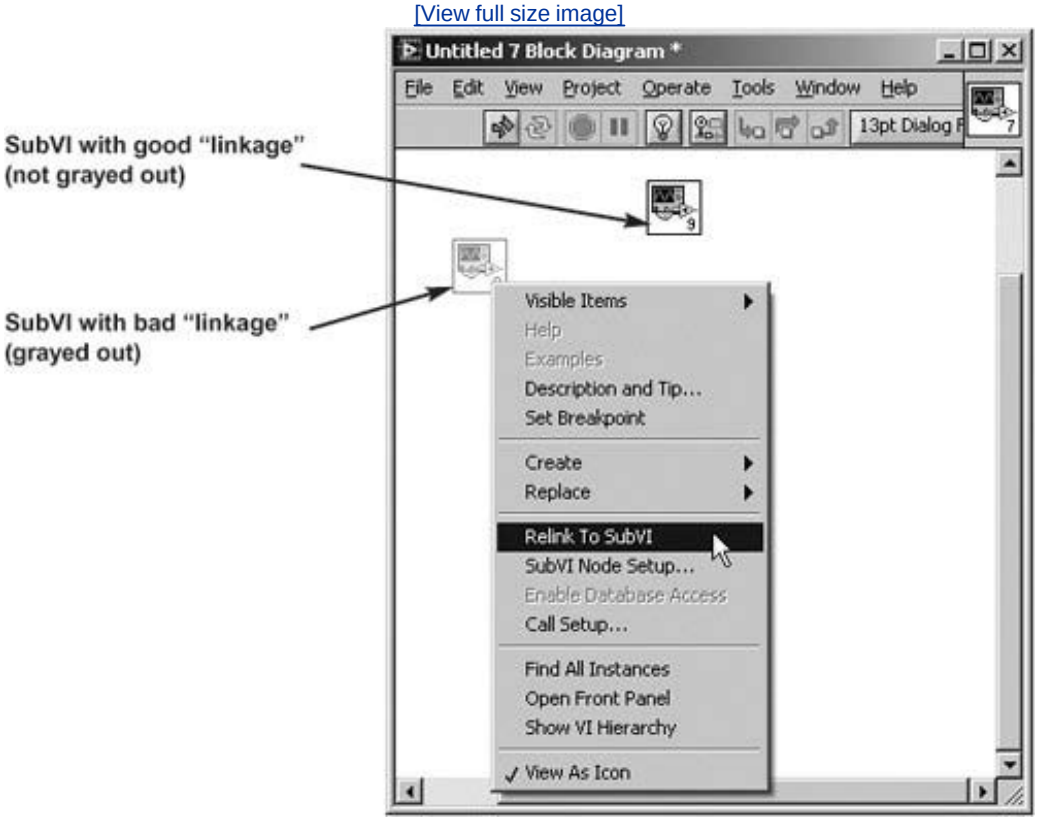
Relink to SubVI: Changing Connector Panes of SubVIs

If you change the connector pane *pattern* (number of terminals, etc.) of a VI that is being used as a subVI, you will have to *relink* the subVI before you will be able to run (if you press the broken Run button of the VI that contains the modified subVI, you will see that there is a "bad linkage to subVI" error with details stating, "The connector pane connections of the subVI do not match the way it is wired on this diagram."). The subVI whose connector pane was modified will be grayed out, indicating that it has bad linkage. You can correct this problem by selecting Relink To SubVI from the grayed-out subVI's pop-up menu, as shown in Figure 5.23.

Figure 5.23. Relinking a subVI with bad linkage (caused by changing the

connector pane pattern)

[View full size image]



Choosing a connector pane pattern that has more terminals than your immediate needs means that you can wire additional controls and indicators to your connector pane, at a later time, without having to relink callers (as you would if you had to change your connector pane pattern). This foresight can save you a lot of time later, and help you avoid potential mistakes that can happen during the relinking process.

PREV NEXT

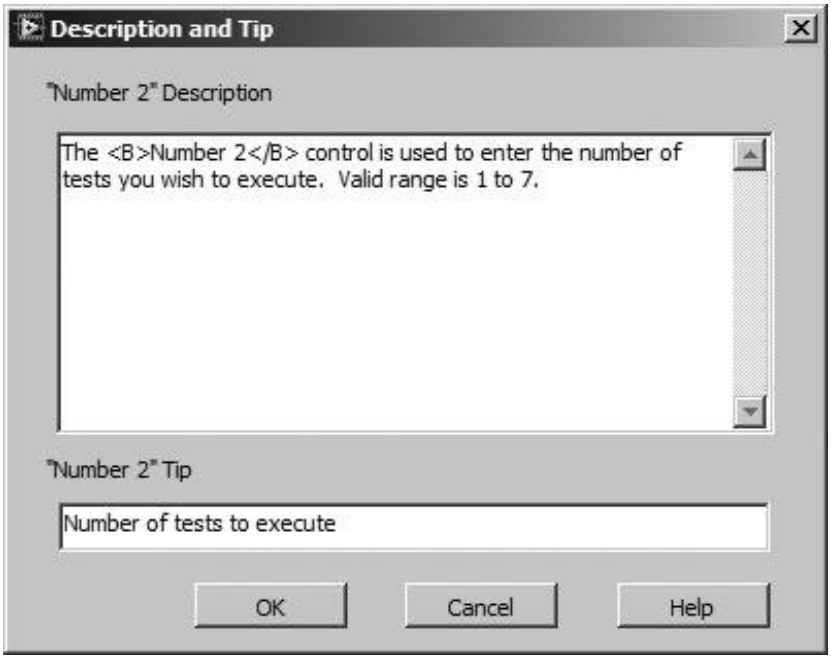
Documenting Your Work

It's important to document your VIs so that others can understand them and so you don't forget why you did something or how it works. This section discusses a few ways you can document your work in LabVIEW.

Creating Descriptions and Tips for Individual Objects

If you want to enter a description of a LabVIEW object, such as a control, indicator, structure, or function, choose Description and Tip . . . from the object's pop-up menu. Enter the description in the resulting Description and Tip dialog box, shown in Figure 5.24, and click OK to save it. You can also enter a tip in the Tip box.

Figure 5.24. Description and Tip dialog



LabVIEW displays the description text in the Help window whenever you pass the cursor over a front panel control or indicator. The tip is displayed as a "tool tip" whenever a cursor pauses over the front panel object if the VI is in run mode, regardless of whether the Help window is open.

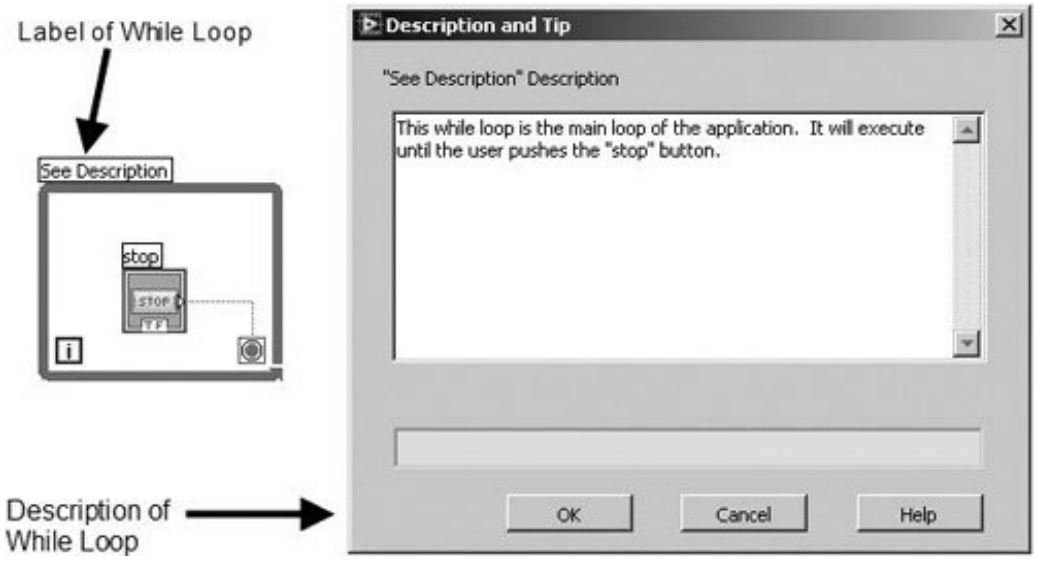
In the Control and VI Descriptions, you can use the HTML bold tags (<B>Text that you want to be bold</B>) to make sections of text bold, in the Context Help window. Currently, bold is the only HTML tag supported.

The best way to set up online help for your VIs is to enter tips and descriptions for all of their controls, indicators, and functions.

For structures and functions on the block diagram, LabVIEW does not show the description text in the Help window when you pass the cursor over the structure or function, as it does for controls and indicators on the front panel. The only way to view the description of the block diagram objects is from the Description and Tip dialog box. Additionally, other programmers will not know to look for descriptions in block diagram objects, so these might go unnoticed if other measures are not taken. For this reason, you might consider changing the label of block diagram objects that have descriptions to state something like "see description," to indicate that there is more information available in the description. Figure 5.25 shows an example of this useful technique.

Figure 5.25. Using a structure's label to indicate that the description contains documentation

[View full size image]

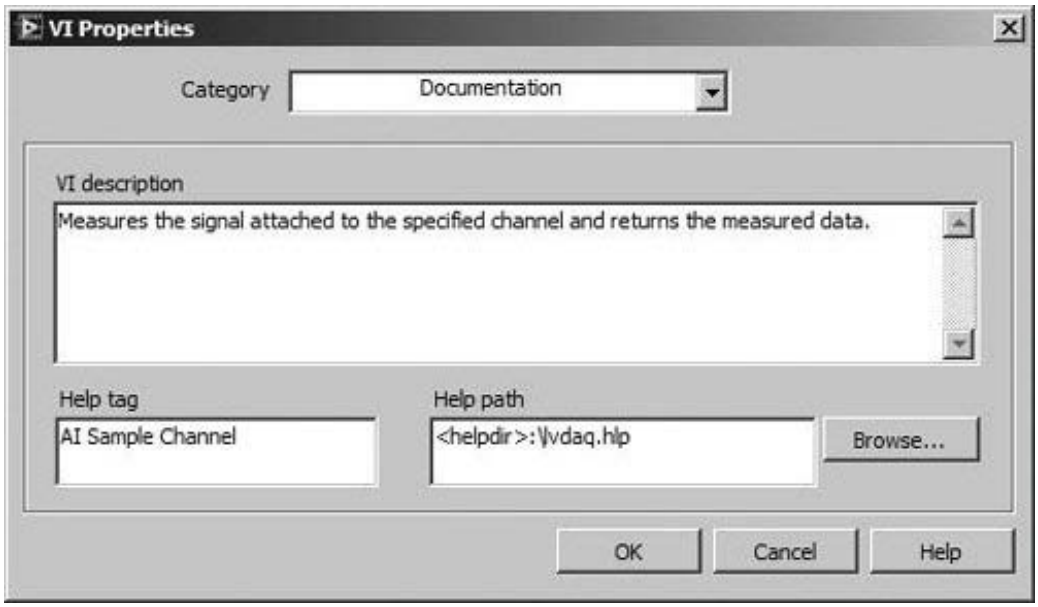


Documenting VIs in the VI Properties

LabVIEW also gives you an easy way to document an entire VI. Selecting VI Properties . . . from the File menu and choosing the Documentation dialog, as shown in Figure 5.26, displays the VI Documentation dialog box for the current VI.

Figure 5.26. VI Properties dialog with Documentation category visible

[View full size image]



You can use the VI Properties dialog box to perform the following functions:

- Enter a description of the VI in the Documentation category. The description area has a scrollbar so that you can edit or view lengthy descriptions. When you use the VI as a subVI, the Help window will display this description when the cursor is over its block diagram icon. You can optionally enter a Help tag and a path to an external Help file.
- See a list of changes made to the VI since you last saved it by pressing the Revision History . . . button in the General category.
- View the path of the VI (i.e., where it is stored) in the General category.
- See how much memory the VI uses in the Memory Category. The memory usage portion of the information box displays the disk and system memory used by the VI. The figure applies only to the amount of memory the VI is using and does not reflect the memory used by any of its subVIs.
- And other interesting things we'll leave for the "Advanced" section. . . .

PREV NEXT

A Little About Printing

LabVIEW has several kinds of printing you can use when you want to make a hard copy of your work:

- You can use the Print Window option from the File menu to make a quick printout of the contents of the current window.
- You can make a comprehensive, custom printout of a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on, by selecting the Print . . . option from the File menu. Choosing this option takes you through a print wizard where you can specify the format you want. You have the option of not only printing a hard copy, but printing to HTML and RTF files as well.
- You can use the LabVIEW programmatic printing features to make VI front panels print under the control of your application. Select Print at Completion from the Operate menu to enable programmatic printing. LabVIEW will then print the contents of the front panel any time the VI finishes executing. If the VI is a subVI, LabVIEW prints when that subVI finishes, before returning to the caller. We'll also talk more about this option in Chapter 14.
- You can have LabVIEW generate custom reports for you as RTF files or HTML files (and even Microsoft Word and Excel reports, with an add-on toolkit from NI). These reports can be printed and can include images and graphs as well as text. We'll look at Report Generation in LabVIEW in more detail in Chapter 16, "Connectivity in LabVIEW."

Activity 5-2: Creating SubVIsPractice Makes Perfect

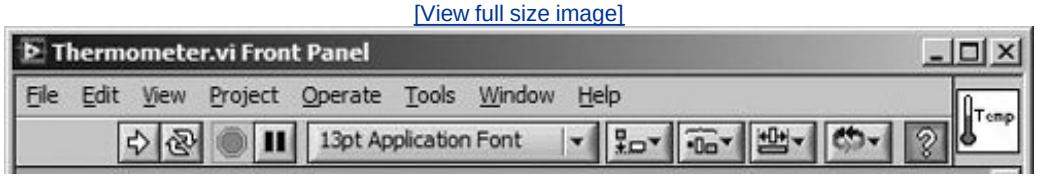
Okay, time to go back to the computer again. You will turn the Thermometer VI you created in the last chapter into a subVI so that you can use it in the diagram of another VI.

1. Open the Thermometer.vi that you created in Activity 4-2. If you saved it in your `MYWORK` directory like we told you to, it should be easy to find. If you are using the sample software or can't find it, use the Thermometer.vi found in `EVERYONE\CH04`.

2. Create an icon for the VI. Open the Icon Editor dialog by selecting Edit Icon . . . from the pop-up menu of the VI's icon pane (on either the front panel or block diagram window). Use the tools described earlier in this chapter (in the "Designing the Icon" section) to create the icon; then click the OK button to return to the main VI. Your icon should appear in the icon pane, as shown in Figure 5.27.

Figure 5.27. The icon pane of Thermometer.vi, after you draw its "Thermometer" icon

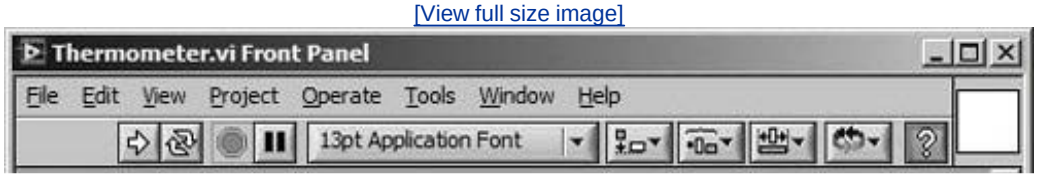
[View full size image]



3. Create the connector by popping up in the icon pane and selecting Show Connector. Because you have only one indicator on your front panel, your connector needs only one terminal (we've mentioned that the default 12-terminal connector has some advantages, but we'll ignore that for now). Pop up on the connector and choose the single terminal (the first pattern, in the upper-left corner) from the Patterns submenu. Your connector should appear as a white box, as shown in Figure 5.28.

Figure 5.28. The connector pane of Thermometer.vi, after you configure it to have only one terminal

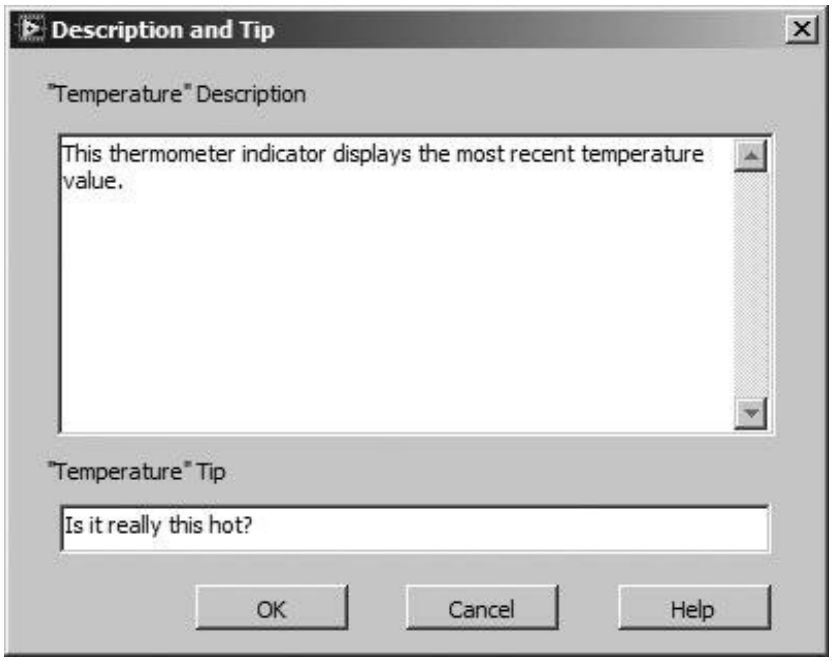
[View full size image]



4. Assign the terminal to the thermometer indicator. Using the Wiring tool (which is probably what the cursor will be automatically), click on the terminal in the connector. The terminal will turn black. Then click on the thermometer indicator. A moving dotted line will frame the indicator. Finally, click in an open area on the panel. The dotted line will disappear and the selected terminal will turn from black to orange, indicating that you have assigned the indicator to that terminal (recall that orange is the color LabVIEW uses for the DBL numeric data type). Pop up and choose Show Icon to return to the icon.

5. Document the `Temperature` indicator by selecting Description and Tip . . . from its pop-up menu. Type in the description and tip as shown and click OK when you're finished (see Figure 5.29).

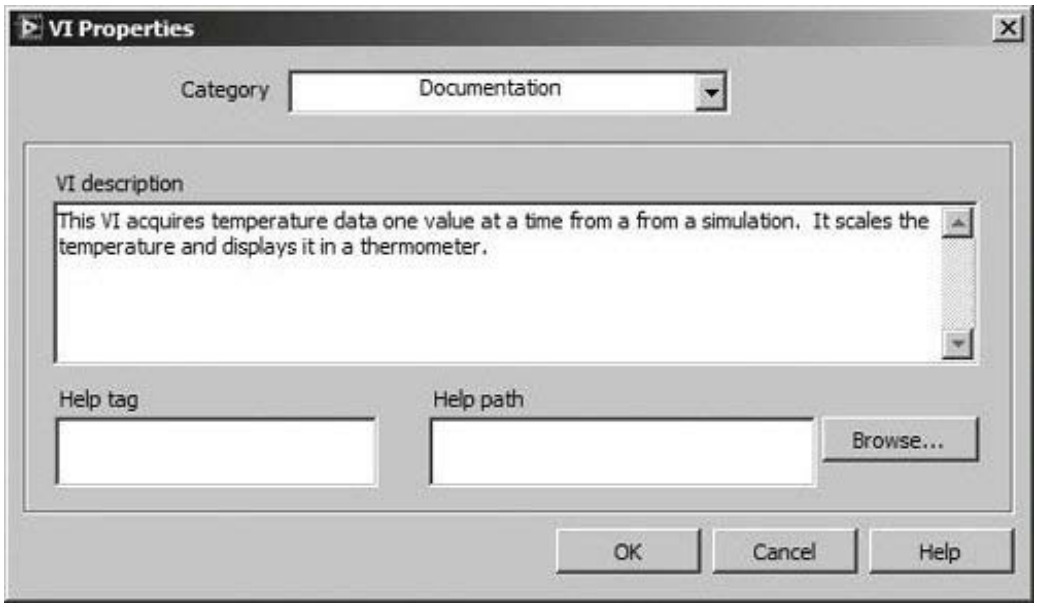
Figure 5.29. The Description and Tip dialog being used to edit the description and tip of the **`Temperature`** indicator



6. Document Thermometer.vi by selecting VI Properties . . . >>Documentation from the File menu or by selecting VI Properties . . . from the icon's pop-up menu, and typing in a description of what it does, as shown in Figure 5.30. Click OK to return to the main VI.

Figure 5.30. Editing the VI description of Thermometer.vi in the VI Properties dialog

[View full size image]



7. Now bring up the Help window by choosing Show Help from the Help menu. When you place the cursor over the icon pane, you will see the VI's description and wiring pattern in the Help window. If your Temperature indicator is not labeled in the VI, it won't have a label in the Help window either.

8. If you have a printer connected to your computer, choose Print Window from the File menu to print the active window. You can decide whether you want to print the front panel or block diagram.

9. Save the changes by selecting Save from the File menu. Excellent work! You will use this VI as a subVI in the next chapter, so make sure to put it in `MYWORK` so you can find it!

10. Just for fun, use the Positioning tool to select the portion of the block diagram shown in Figure 5.31; then choose Create SubVI from the Edit menu to automatically turn it into a subVI. Notice that the `Temperature` indicator remains part of the caller VI, as shown in Figure 5.32. Double-click on the new subVI to see its front panel.

Figure 5.31. Selected portion of the block diagram

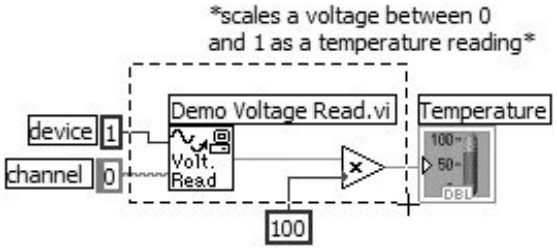
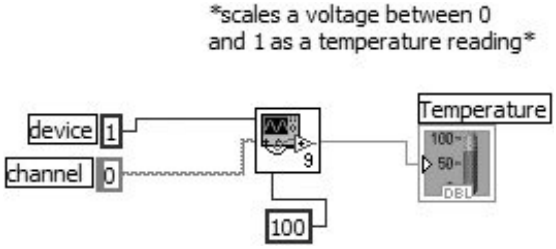


Figure 5.32. Created subVI



11. Close both the new subVI and Thermometer.vi. This time, do not save any changes.

Frequently, people create an icon for a VI and forget about the connector. If you haven't assigned the connector, you will be unable to wire inputs and outputs to your VI when you try to use it as a subVI and you may find the source of your wiring inability very difficult to locate.

PREV NEXT

Wrap It Up!

LabVIEW offers several ways to save your VIs. Saving them in LLB files, which are special LabVIEW files containing groups of VIs, is not recommended. Regardless of how you save, be sure to do it frequently!

You can take advantage of LabVIEW's many useful debugging features if your VIs do not work right away. You can *single-step* through your diagram node by node, animate the diagram using *execution highlighting*, and suspend subVIs when they are called so that you can look at input and output values by setting a *breakpoint*. You can also use the *probe* to display the value a wire is carrying at any time. Each of these features allow you to take a closer look at your problem.

SubVIs are the LabVIEW equivalent of subroutines. All subVIs must have an icon and a connector. You can create subVIs from existing VIs or from a selected part of a block diagram. To prevent wiring mistakes with subVIs, use their online help and specify their inputs as *required*, *recommended*, *or optional*. Import an existing subVI into a calling VI's block diagram by choosing Select a VI . . . from the Functions palette, and then selecting the subVI you want from the dialog box. SubVIs represent one of LabVIEW's most powerful features. You will find it very easy to develop and debug reusable, low-level subVIs and then call them from higher-level VIs.

As with all programming, it is a good idea to document your work in LabVIEW. You can document an entire VI by entering a description under the Documentation dialog of the VI Properties . . . from the File menu. This description is also visible in the Help window if you pass the cursor over the VI's icon. You can document individual front panel objects and block diagram functions by selecting Description and Tip . . . from the pop-up menu and then entering your text in the resulting dialog box. Descriptions for front panel objects will also appear in the Help window when you pass the cursor over the object.

LabVIEW offers several options for printing VIs; you can print the active window, specify what parts of a VI you want to print (such as front panel, block diagram, or subVI information), or set the VI to print programmatically.

Congratulations! You've now covered LabVIEW's fundamental operations. Now you're ready to learn about some of LabVIEW's powerful structures and functions, and how to write cool programs with them.

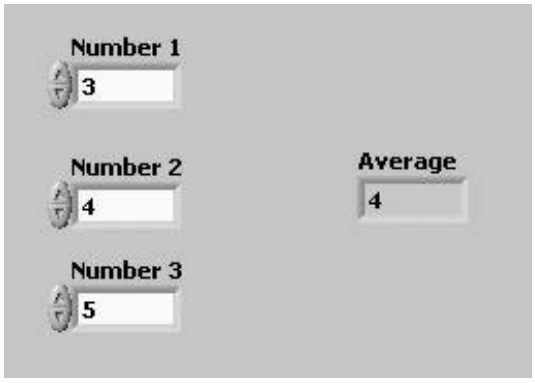
Additional Activities

Here's some more practice for you. If you get really stuck, look in `EVERYONE\CH05` for the solutions.

Activity 5-3: Find the Average

Create a subVI that averages three input numbers and outputs the result (see Figure 5.33). Remember to create both the icon and connector for the subVI. Save the VI as Find the Average.vi.

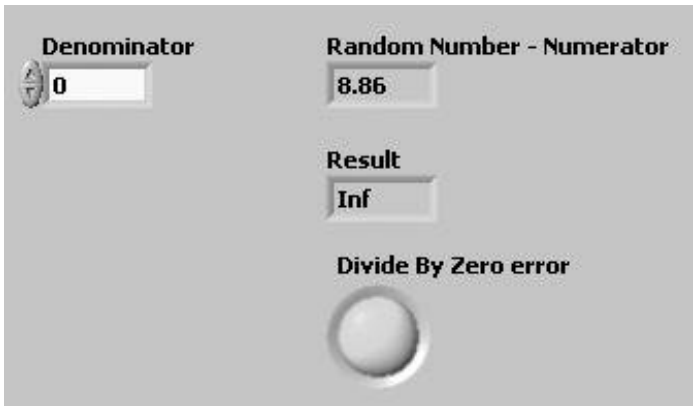
Figure 5.33. Find the Average.vi front panel



Activity 5-4: Divide by Zero (Who Says You Can't?)

Build a VI that generates a random number between zero and ten, and then divides it by an input number and displays the result on the front panel (see Figure 5.34). If the input number is zero, the VI lights an LED to flag a "divide by zero" error. Save the VI as Divide by Zero.vi.

Figure 5.34. Divide by Zero.vi front panel



Use the Equal? Function found in the Comparison subpalette of the Functions palette.

PREV NEXT

6. Controlling Program Execution with Structures

Overview

Structures, an important type of node, govern execution flow in a VI, just as control structures do in a standard programming language. This chapter introduces you to the main structures in LabVIEW: the While Loop, the For Loop, the Case Structure, and the Sequence Structure, and then shows you how to combine the While Loop with the Case Structure into a powerful application framework. You will also learn how to implement lengthy formulas using the Formula Node and simple formulas with the Expression Node, how to pop up a dialog box containing your very own message, and a few basics on how to control the timing of your programs, as well as how the Timed Structures provide advanced timing and synchronization. You might want to take a look at some examples of structures in `EXAMPLES\GENERAL\STRUCTS.LLB` *in LabVIEW*.

Goals

- Know the uses of the While Loop and the For Loop and understand the differences between them
- Recognize the necessity of shift registers in graphical programming
- Understand the different types of Case Structuresnumeric, string, and Boolean
- Learn how to regulate execution order using Sequence Structures
- Understand the dangers of the Stacked Sequence Structure and Sequence Locals
- Use the Formula Node to implement long mathematical formulas
- Make LabVIEW pop up a dialog box that says anything you tell it to
- Understand how to use some of LabVIEW's simple timing functions
- Use the Expression Node to implement formulas with only one variable
- Understand how the Timed Structures and VIs allow Timed Loops and Timed Sequences to be synchronized
- Learn how to combine the While Loop with the Case Structure to build simple, yet powerful and scalable, application frameworks

Key Terms

- For Loop
- While Loop
- Iteration terminal
- Conditional terminal
- Count terminal
- Tunnel
- Coercion dot
- Shift register
- Select function
- Case Structure
- Selector terminal
- Dialog box
- Wait (ms)
- Flat Sequence Structure
- Stacked Sequence Structure
- Formula Node
- Expression Node
- Timed Structure

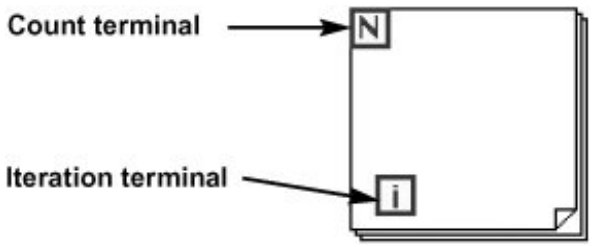
Two Loops

If you've ever programmed in any language, you've probably wanted to repeat the execution of a section of code. LabVIEW offers two loop structures to make this easy. You can use the *For Loop* or *While Loop* to control repetitive operations in a VI. A For Loop executes a specified number of times; a While Loop executes until a specified condition is true (or false, depending on how the While Loop is configured). You can find both loops under the Programming>>Structures subpalette of the Functions palette.

The For Loop

A *For Loop* executes the code inside its borders, called its *subdiagram*, for a total of *count* times, where the *count* equals the value contained in the *count terminal*. You can set the count by wiring a value from outside the loop to the count terminal. If you wire 0 to the count terminal, the loop does not execute.

Figure 6.1. For Loop



The *iteration terminal* contains the current number of completed loop iterations: 0 during the first iteration, 1 during the second, and so on, up to N-1 (where N is the number of times you want the loop to execute).

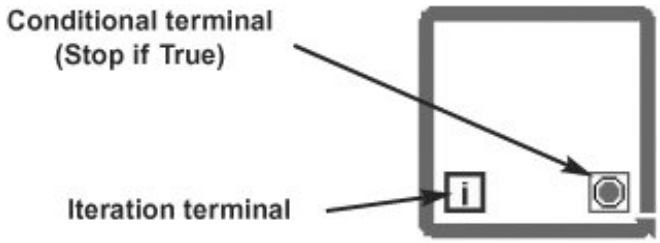
The For Loop is equivalent to the following pseudocode:

```
for i = 0 to N-1
      Execute subdiagram
```

The While Loop

The *While Loop* executes the subdiagram inside its borders until the Boolean value wired to its *conditional terminal* is TRUE (meaning "yes, stop the loop"), as shown in Figure 6.2. LabVIEW checks the conditional terminal value at the *end* of each iteration. If the value is FALSE (meaning "no, don't stop the loop"), another iteration occurs.

Figure 6.2. While Loop



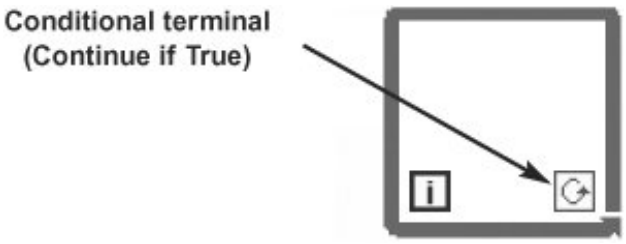
The While Loop's *iteration terminal* behaves exactly like the one in the For Loop.

The While Loop is equivalent to the following pseudocode:

```
Do
      Execute subdiagram
While condition is FALSE
(Until condition is TRUE)
```

You can also change the state that the conditional terminal of the While Loop checks, so that instead of looping *while false*, you can have it loop *while true*. To do this, you pop up on the conditional terminal, and select "Continue if True." The While Loop will look like Figure 6.3.

Figure 6.3. While Loop with "Continue if True" conditional terminal



The While Loop in Figure 6.3 is equivalent to the following pseudocode:

```
Do
      Execute subdiagram
While condition is NOT TRUE
```

As of LabVIEW 7.0, the initial behavior of the While Loop's conditional terminal (when first dropped onto the block diagram of a VI) is Stop if True ( ). However, in LabVIEW 6.1 and earlier, the default state of the conditional terminal is Continue if True ( ). When you are following along with the examples in this book, make sure that you have correctly configured any While Loop conditional terminals to match those in the examples.

If you want to convert a structure to a different type (for example, turn a For Loop into a While Loop), you can right-click on the outer edge of the structure and select a Replace item from the shortcut menu. You can only replace a structure with similar structures. Experiment with this feature to see how it works.

Placing Objects Inside Structures

For Loop Cursor

While Loop Cursor

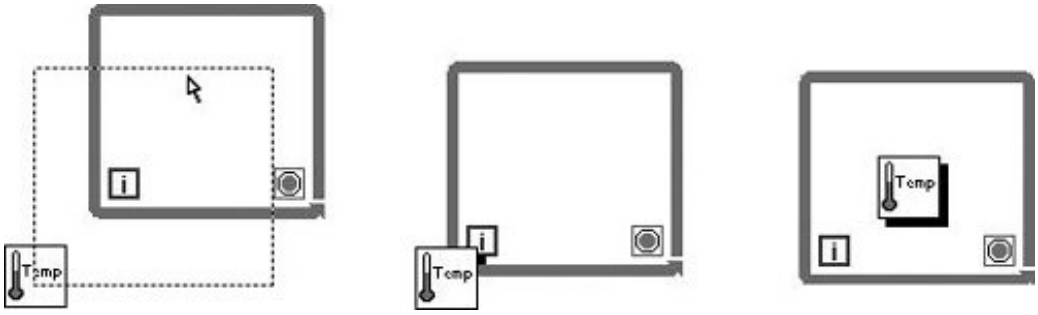
When you first select a structure from the Programming>>Structures subpalette of the Functions palette, the cursor appears as a miniature of the structure you've selected; for example, the For Loop or the While Loop. You can then click where you want one corner of your structure to be, and drag to define the borders of your structure. When you release the mouse button, the structure will appear, containing all objects you captured in the borders.

Once you have the structure on the diagram, you can place other objects inside either by dragging them in, or by placing them inside when you select them from the Functions palette. To make it clear to you that you are dragging something *into* a structure, the *structure's* border will highlight as the object moves inside. When you drag an object *out* of a structure, the *block diagram's* border (or that of an outer structure) will highlight as the object moves outside.

You can resize an existing structure by grabbing and dragging a resizing handle on an edge with the Positioning tool.

If you move an existing structure so that it overlaps another object, the overlapped object will be visible above the edge of the structure. If you drag an existing structure completely over another object, that object will display a thick shadow to warn you that the object is *over* or *under* rather than *inside* the structure. Both of these situations are shown in Figure 6.4.

Figure 6.4. A subVI that is not inside a structure, floating above it and hiding beneath it



Auto Grow

All structures have a property called *Auto Grow*, which can be toggled by checking or unchecking the Auto Grow item of the structure's pop-up menu. When enabled, this feature causes two very useful

behaviors. First, it causes the structure to grow automatically when you move objects inside the structure, to positions that overlap the structure's border. Second, this feature prevents you from resizing a structure to a size smaller than the objects inside it. Both of these features prevent objects inside the structure from being hidden inside the structure but outside its frame.

You can change whether LabVIEW enables this setting for new structures placed onto the block diagram by changing the *Place structures with Auto Grow enabled* checkbox in the Block Diagram section of the Tools>>Options . . . dialog.

Removing Structures: Don't Just Delete Them

Whenever you want to remove a structure, be careful. If you just delete that While Loop, you'll also delete all the objects inside of itfunctions, subVIs, wires, and all.

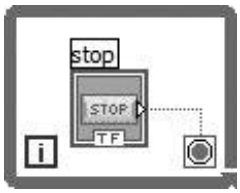
Unless you want to delete all the code inside a structure, you should pop up on the structure and select "Remove While Loop" (for While Loops) or "Remove For Loop" (for For Loops). This will remove the structure, but leave all the other code on the block diagram. It will also leave all wires intact that pass through the frame of the structure via tunnels and shift registers.

Terminals Inside Loops and Other Behavioral Issues

Data passes into and out of a loop through a little box on the loop border called a *tunnel*. Because LabVIEW operates according to dataflow principles, inputs to a loop must pass their data in before the loop executes. *Loop outputs pass data out only after the loop completes all iterations*.

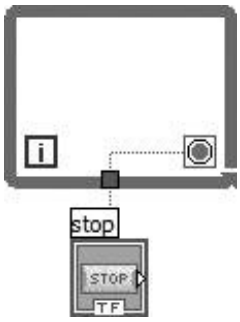
Also, according to dataflow, *you must place a terminal inside a loop if you want that terminal checked or updated on each loop iteration.* For example, the While Loop in Figure 6.5 checks its Boolean control each time it loops. When the loop reads a TRUE value, it terminates.

Figure 6.5. Stop button inside a loop (reads the value, every loop iteration)



If you place the terminal of the Boolean control outside the While Loop, as pictured in the right loop of Figure 6.6, you create an infinite loop or a loop that executes only once, depending on the Boolean's initial value. True to dataflow, LabVIEW reads the value of the Boolean *before* it enters the loop, not within the loop or after completion.

Figure 6.6. Stop button outside a loop (reads the value only once, before the loop executes)



Similarly, the `Digital Indicator` in the loop in Figure 6.7 will update during each loop iteration. The `Digital Indicator` in the loop in Figure 6.8 will update only once, after the loop completes. It will contain the random number value from the last loop iteration.

Figure 6.7. An indicator inside a loop (updated each iteration)

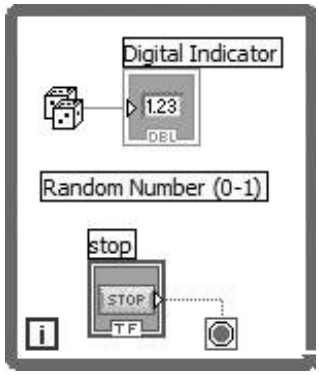
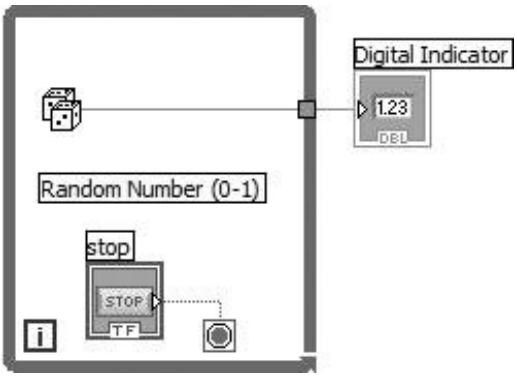


Figure 6.8. An indicator outside a loop (updated only once, on loop completion)



Remember, the first time through a For Loop or a While Loop, the iteration count is zero! If you want to show how many times the loop has actually executed, you must add one to the count!

Activity 6-1: Counting the Loops

In this activity, you get to build a For Loop that displays its count in a chart on the front panel. You will choose the `Number of Iterations`, and the loop will count from zero up to that number minus one (because the iteration count starts at zero). You will then build a While Loop that counts until you stop it with a Boolean switch. Just for fun (and also to illustrate an important point), you will observe the effect of putting controls and indicators outside the While Loop.

1. Create a new panel by selecting New VI from the File menu or by clicking the New VI text in the LabVIEW Getting Started dialog box.

2. Build the front panel and block diagram shown in Figures 6.9 and 6.10. The For Loop is located in the Programming>>Structures subpalette of the Functions palette. You might use the Tile Left and Right command from the Windows menu so that you can see both the front panel and the block diagram at the same time.

Figure 6.9. The front panel of the VI you will build during this activity

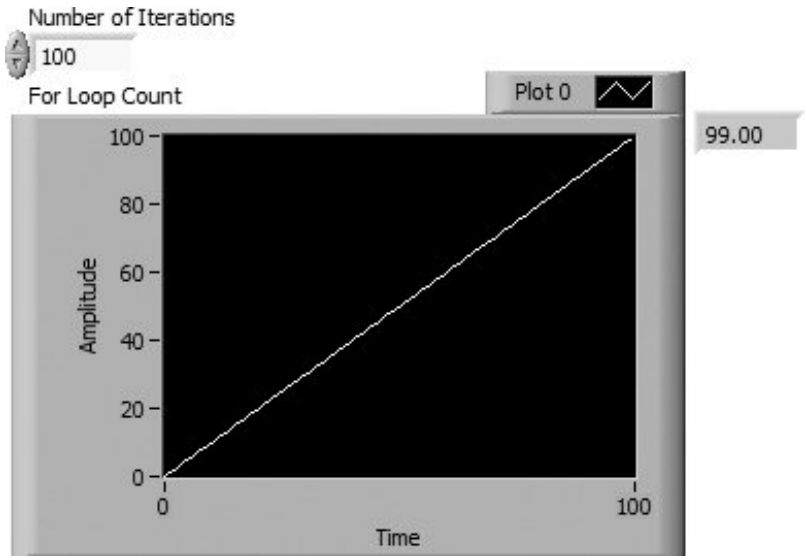
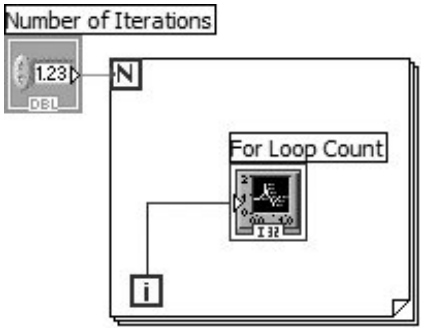


Figure 6.10. The block diagram of the VI you will build during this activity



Drop a Waveform Chart from the Modern>>Graph subpalette of the Controls palette onto your front panel. Label it `For Loop Count`. We'll talk more about charts and graphs in Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs." Use a digital control from the Programming>>Numeric subpalette for your `Number of Iterations` control.

3. Pop up on the Waveform Chart and select AutoScale Y from the Y Scale pull-out menu so that your chart will scale to fit the For Loop count. Then pop up on the chart and Visible Items>>Digital Display. Input a number to your `Number of Iterations` control and run the VI. Notice that the digital indicator counts from 0 to N-1, NOT 1 to N (where N is the number you specified)! Each time the loop executes, the chart plots the For Loop count on the Y axis against time on the X axis. In this case, each unit of time represents one loop iteration.

4. Notice the little gray dot present at the junction of the count terminal and the `Number of Iterations` wire. It's called a coercion dot, and we'll talk about it after this exercise. Pop up on the `Number of Iterations` control and choose I32 Long from the subpalette to make it go away.

5. You can save the VI if you want, but we won't be using it again. Open up another new window so you can try out the While Loop.

6. Build the VI shown in Figures 6.11 and 6.12. Remember, Booleans appear on the front panel in their default FALSE position. Also, remember to set the While Loop's conditional terminal to Continue if True.

Figure 6.11. The front panel of the VI you will build

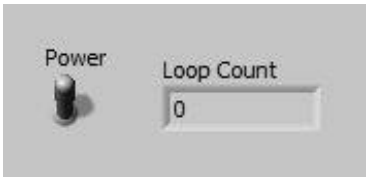
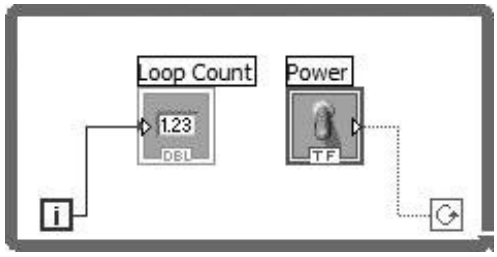


Figure 6.12. The block diagram of the VI you will build



7. Flip the switch up to its TRUE position by clicking on it with the Operating tool and run the VI. When you want to stop, click on the switch to flip it down to FALSE. `Loop Count` will update during each loop iteration.

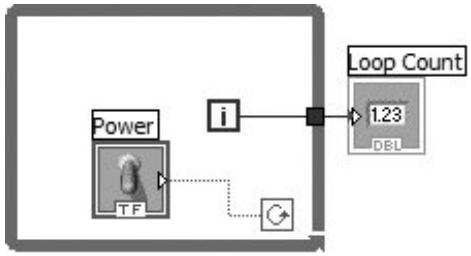

Operating Tool

8. With the switch still in the FALSE position, run the VI again. Notice that the While Loop executes once, but only once. Remember, the loop checks the conditional terminal at the *end* of an iteration, so it always executes at least once, regardless of what value is wired to it.

9. Now go to the block diagram and move the `Loop Count` indicator outside the loop as shown in

Figure 6.13. You will have to rewire the indicator; the tunnel is created automatically as the wire leaves the loop.

Figure 6.13. Your VI, after moving the **Loop Count** indicator outside the While Loop



10. Make sure the switch is TRUE and run the VI. Notice that the indicator updates only after you flip the switch FALSE and the loop has finished executing; it contains the final value of the iteration count, which is passed out after the loop completes. You will learn more about passing data out of loops in Chapter 7, "LabVIEW's Composite Data: Arrays and Clusters." *Until then, do not try to pass scalar data out of a For Loop like you just did in a While Loop, or you will get bad wires and you won't understand why.* It can easily be done, but you will have to learn a little about auto-indexing first.

11. Save the VI. Place it in your `MYWORK` directory and call it `Loop Count.vi`.

12. Now, just to demonstrate what *not* to do, drag the switch out of the loop (but leave it wired). Make sure the switch is TRUE, run the VI, and then hit the switch to stop it. It won't stop, will it? Once LabVIEW enters the loop, it will not check the value of controls outside of the loop (just like it didn't update the `Loop Count` indicator until the loop completed). Go ahead and hit the Abort button on the Toolbar to halt execution. If your switch had been FALSE when you started the loop, the loop would have only executed once instead of forever. Close the VI and do not save changes.


Abort Button

The Coercion Dot

Remember the little gray dot present at the junction of the For Loop's count terminal and the `Number of Iterations` wire in the last activity? It's the *coercion dot*, so named because LabVIEW is coercing one numeric representation to fit another. If you wire two terminals of different numeric representations together, LabVIEW converts one to the same representation as the other. In the previous exercise, the count terminal has a 32-bit integer representation, while the `Number of Iterations` control is by default a double-precision floating-point number until you change it. In this

case, LabVIEW converts the double-precision floating-point number to a long integer. In doing so, LabVIEW makes a new copy of the number in memory, in the proper representation. This copy takes up space. Although the extra space is negligible for scalar numbers (single-valued data types), it can add up quickly if you are using arrays (which store multiple values). Try to minimize the appearance of the coercion dot on large arrays by changing the representation of your controls and indicators to exactly match the representation of data they will carry.

To make it easier to identify coercion dots on the block diagram (which encourages you to follow the preceding rule), you can change their color from the default gray, to a more noticeable color such as bright red. To change the color, open the Tools>>Options . . . dialog and uncheck the Use default colors option (in the Colors category). Click on the Coercion Dots color and change it to any color you like, from the color palette that appears.

When a VI converts floating-point numbers to integers, it rounds to the nearest integer. A number with a decimal value of ".5" is rounded to the nearest even integer.

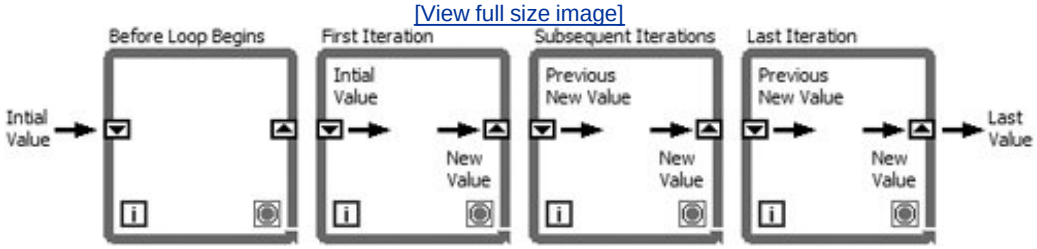
An easy way to create a count terminal input with the correct data type and representation is to pop up on the count terminal and select Create Constant (for a block diagram constant) or Create Control (for a front panel control). Likewise, you can create indicators in a similar manorfor example, you can pop up on the iteration terminal and select Create Indicator (for a front panel indicator) to view the iteration count as the loop executes.

Shift Registers

Shift registers, available for While Loops and For Loops, are a special type of variable used to transfer values from one iteration of a loop to the next (see Figure 6.14). They are unique to and necessary for LabVIEW's graphical structure; we'll talk more about their uses in a little while. You create a shift register by popping up on the left or right loop border and selecting Add Shift Register from the pop-up menu.

Figure 6.14. Shift registers

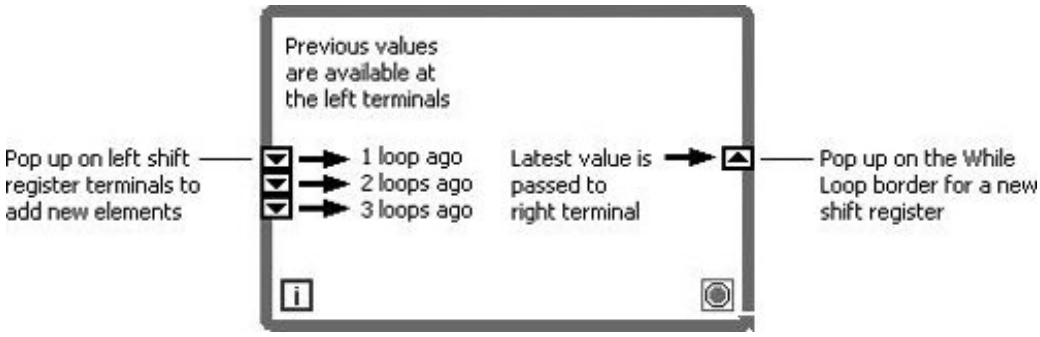
[View full size image]



You can configure the shift register to remember values from several previous iterations, as shown in Figure 6.15, a useful feature when you are averaging data values acquired in different iterations. To access values from previous iterations, create additional terminals by popping up on the *left* terminal and choosing Add Element from the pop-up menu. Alternatively, you can create additional terminal elements by hovering over the Shift Register and dragging the grab handles that appear.

Figure 6.15. A While Loop having one shift register with multiple left terminal elements

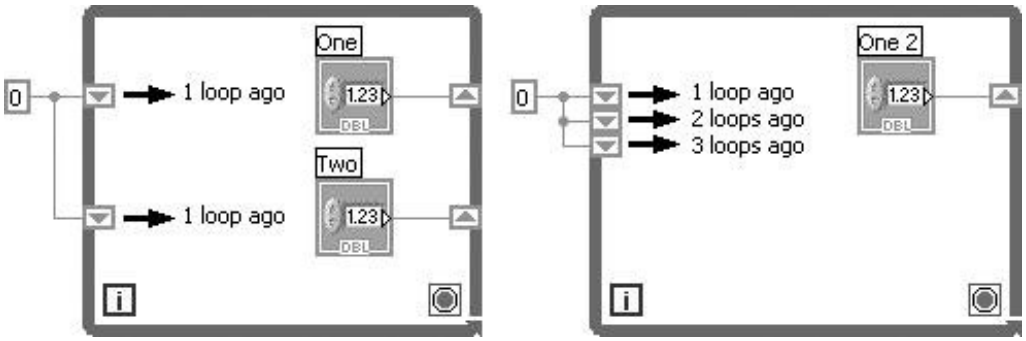
[View full size image]



You can have many different shift registers storing many different variables on the same loop. Just pop up on the loop border and add them until you have as many pairs as you need. The left terminal will always stay parallel with its right terminal; if you move one, they both move. So if you have a lot of shift registers on your loop and can't tell exactly which ones are parallel, just select one and its partner will be automatically selected, or move one terminal a little and watch its mate follow.

Don't make the common mistake of confusing multiple variables stored in multiple shift registers with a single variable stored from multiple previous iterations in one shift register. Figure 6.16 shows the difference.

Figure 6.16. (Left) Two separate variables. (Right) Several loop values of one variable.

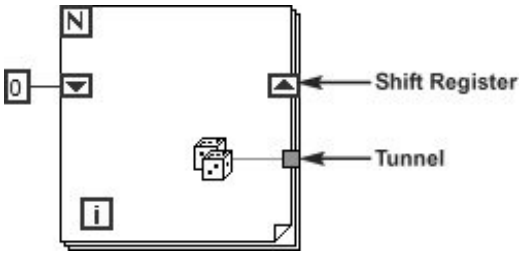


If you're still a little confused, don't worry. Shift registers are a completely new and different concept, unlike anything you may have encountered in a traditional programming language. Stepping through the next exercise should demonstrate them more clearly for you.

Make sure to wire directly to the shift register terminal so that you don't accidentally create

an unrelated tunnel into or out of the loop. Figure 6.17 shows the difference between a shift register and a tunnel.

Figure 6.17. Shift register and tunnel, two different ways of passing data through the walls of a loop

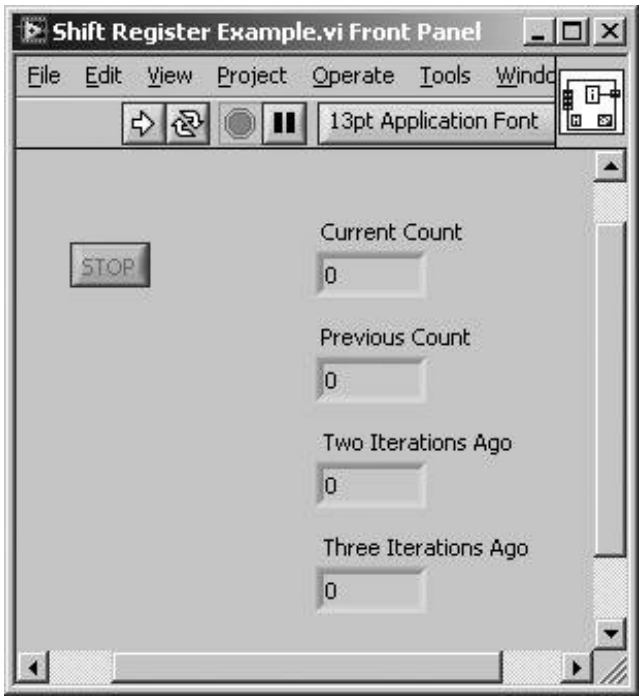


Activity 6-2: Shift Register Example

To give you an idea of how shift registers work, you will observe their use to access values from previous iterations of a loop. In this VI, you will be retrieving count values from previous loops.

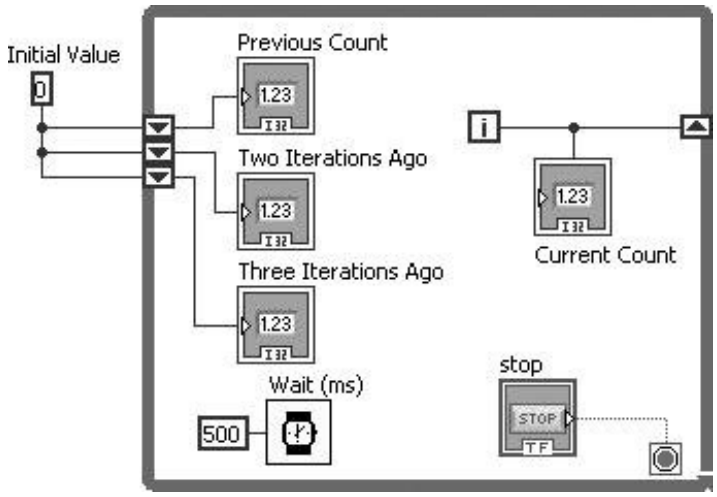
1. Open Shift Register Example.vi, located in `EVERYONE\CH06` (see Figure 6.18).

Figure 6.18. Shift Register Example.vi front panel



The front panel has four digital indicators. The `Current Count` indicator will display the current value of the loop count (it is wired to the iteration terminal). The `Previous Count` indicator will display the value of the loop count one iteration ago. The `Two Iterations Ago` indicator will display the value from two iterations ago, and so on (see Figure 6.19).

Figure 6.19. Shift Register Example.vi block diagram



2. Open the block diagram window by choosing Show Diagram from the Windows menu.

   The zero wired to the left shift register terminals initializes the elements of the shift register to zero. At the beginning of the next iteration, the old `Current Count` value will shift to the top left

terminal to become `Previous Count`. `Previous Count` shifts down into `Two Iterations Ago`, and so on. The timer function Wait (ms) tells the loop to wait 500 ms before iterating.

3. After examining the block diagram, show both the panel and the diagram at the same time by choosing Tile Left and Right from the Windows menu.

4. Enable the execution highlighting by clicking on the Execution Highlighting button.


Execution Highlighting Button

5. Run the VI and carefully watch the bubbles. If the bubbles are moving too fast, stop the VI and click on the Step Into button to put the VI in single-step mode. Click on the button again to execute each step of the VI. Watch how the front panel indicator values change.


Step Into Button

Notice that in each iteration of the While Loop, the VI "funnels" the previous values through the left terminals of the shift register using a first in, first out (FIFO) algorithm. Each iteration of the loop increments the count terminal wired to the right shift register terminal, `Current Count`, of the shift register. This value shifts to the left terminal, `Previous Count`, at the beginning of the next iteration. The rest of the shift register values at the left terminal funnel downward through the terminals. In this example, the VI retains only the last three values. To retain more values, add more elements to the left terminal of the shift register by popping up on it and selecting Add Element.

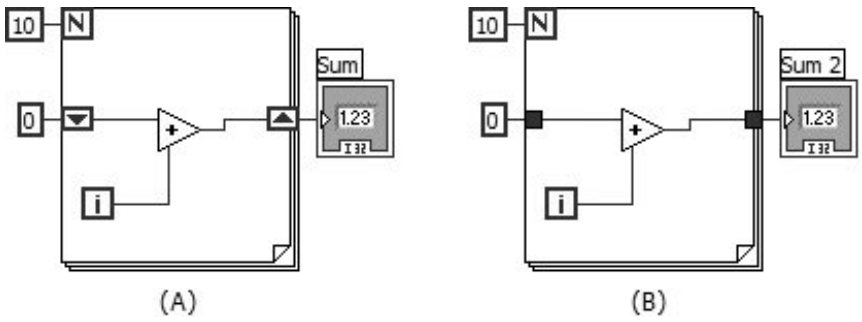
Stop the VI by pressing the `STOP` button on the front panel. If you are in single-step mode, keep pressing the step button until it completes.

6. Close the VI. Do not save any changes. Another job well done!

Why You Need Shift Registers

Observe the example illustrated in Figure 6.20. In loop (A), you are creating a running sum of the iteration count. Each time through the loop, the new sum is saved in the shift register. At the end of the loop, the total sum of 45 is passed out to the numeric indicator. In loop (B), you have no shift registers, so you cannot save values between iterations. Instead, you add zero to the current "i" each time, and only the last value of 9 will be passed out of the loop.

Figure 6.20. Two loops showing the difference between shift registers (A) and tunnels (B)

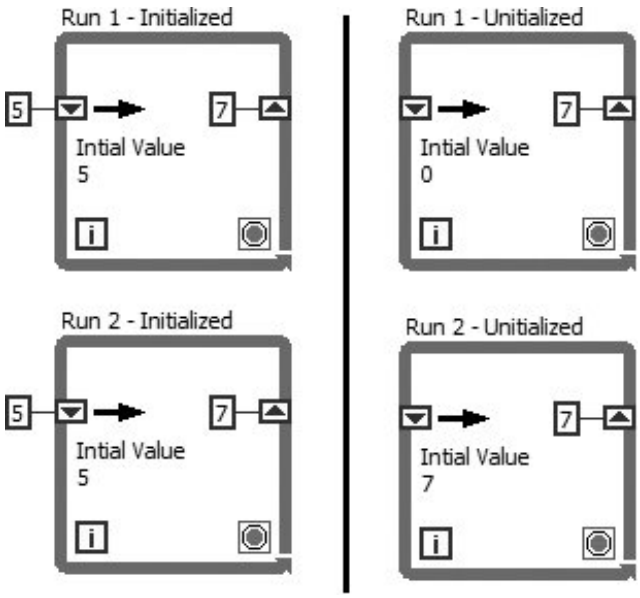


Or what about a case where you need to average values from successive loop iterations? Maybe you want to take a temperature reading once per second, and then average those values over an hour. Given LabVIEW's graphical nature, how could you wire a value produced in one loop iteration into the next iteration without using a shift register?

Initializing Shift Registers

To avoid unforeseen and possibly nasty behavior, you should *always initialize your shift registers* unless you have a specific reason not to and make a conscious decision to that effect. To initialize the shift register with a specific value, wire that value to all the left terminals of the shift register from outside the loop, as shown in the left two loops in Figure 6.21. If you do not initialize it, the initial value will be the default value for the shift register data type the first time you run your program. In subsequent runs, the shift register will contain whatever values are left over from previous runs.

Figure 6.21. Two loops (left and right) on subsequent iterations (top and bottom) showing the effect of uninitialized shift registers (right) vs. initialized shift registers (left)



For example, if the shift register data type is Boolean, the initial value will be FALSE for the first run. Similarly, if the shift register data type is numeric, the initial value will be zero. The second time you run your VI, an uninitialized shift register will contain values left over from the first run! Study Figure 6.21 to make sure you understand what initialization does. The two loops in the left column show what happens when you run a program that contains an initialized shift register twice. The right column shows what happens if you run a program containing an uninitialized shift register two times. Note the initial values of the shift registers in the two bottom loops.

LabVIEW does not discard values stored in the shift register until you close the VI and remove it from memory. In other words, if you run a VI containing uninitialized shift registers, the initial values for the subsequent run will be the ones left over from the previous run. You seldom want this behavior (although there are some advanced uses such as the "Functional Global," which will be discussed in Appendix D, "LabVIEW Object-Oriented Programming") and the resulting problems can be very difficult to spot!

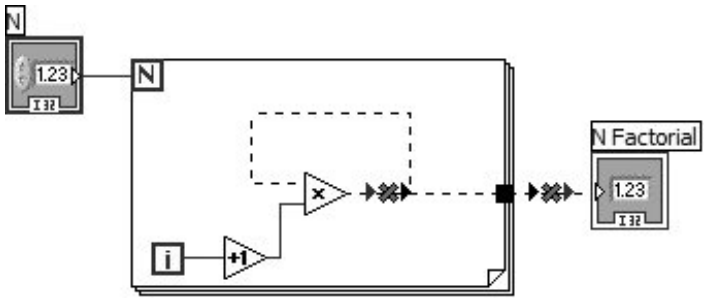
If you resize a shift register to show multiple previous iterations (on the left shift register),

LabVIEW requires you to initialize all the shift register elements if any shift register elements are initialized. Otherwise, your VI will be broken (unable to run).

The Feedback Node

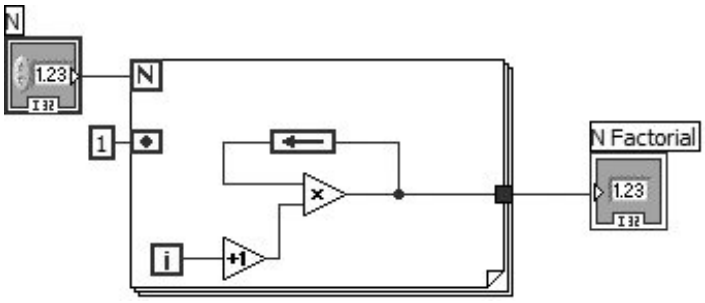
Generally, LabVIEW does not allow you to create a "cycle," where the output of block of code is used as the input for the same block of codeand attempting to do so will cause wires to become broken. You can see an example of a "cycle" in Figure 6.22. The cycle is broken due to the data flow rules, which state that (1) a node cannot execute until data flows into all of its input terminals and that (2) data does not flow out of a node's output terminals until the node finishes executing. Because a "cycle" uses the output terminals as a source for the input terminals, data can never flow into the input terminals because data will never flow out of the output terminals, because the node will not run until data flows into the input terminalsquite a paradox.

Figure 6.22. A cycle, which is broken due to an input depending on an output



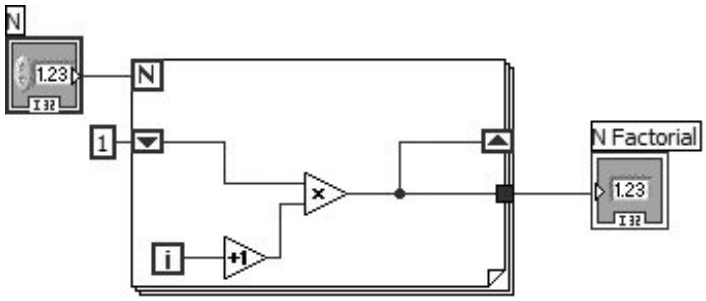
However, if we are inside a While Loop or For Loop, we can place a Feedback Node in between the output terminal and the input terminal, and our code will now work. You can see an example of this in Figure 6.23. In fact, by default, LabVIEW will automatically place a Feedback Node in your cycle, when it is created, if you are inside a While Loop or For Loop. This setting is controlled by the Auto-insert Feedback Nodes in cycles checkbox in the Block Diagram section of the LabVIEW options dialog.

Figure 6.23. A feedback node allows the cycle to work



To understand exactly how the Feedback Node works, we should realize that a Feedback Node is really just a Shift Register in disguise. In fact, the code in Figure 6.23 is equivalent to the code in Figure 6.24. The first thing to note is that the arrow on the Feedback Node indicates the direction of data flow. The input terminal of the Feedback Node is equivalent to the right terminal of a Shift Register, and the output terminal of the Feedback node is equivalent to the left terminal of a Shift Register. For initializing a Feedback Node, we use the Initializer Terminal, which can be shown or hidden by right-clicking on the Feedback Node and selecting or deselecting the Initializer Terminal option. You can easily convert a Feedback Node to a Shift Register, and vice versa; simply right-click on the Feedback Node and select Replace with Shift Register or right-click on the Shift Register and select Replace with Feedback Node.

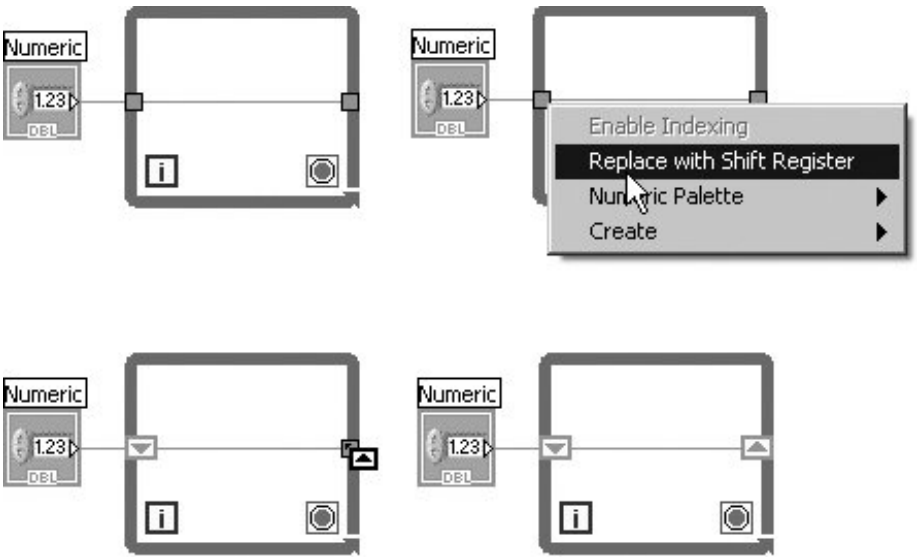
Figure 6.24. A shift register created from a cycle (using the Replace with Shift Register feature)



Converting Tunnels to Shift Registers (and Vice Versa)

You may realize, in the process of writing your code, that a tunnel should in fact be replaced by a shift register. Fortunately, LabVIEW makes this very easysimply right-click on the tunnel and select Replace with Shift Register from the pop-up menu (Figure 6.25).

Figure 6.25. Replacing input and output tunnels with a shift register



Shift Register Cursor

The first thing you will notice is that the tunnel is replaced by a shift register. Also, you will notice that the mouse cursor now appears as a Shift Register Cursor. Use this cursor to select (mouse-click) another tunnel that you wish to convert to the opposing side of the shift register, or click anywhere on or inside the loop to place the opposing side of the shift register without converting another tunnel (see Figure 6.25).

To convert a shift register into tunnels, simply right-click on the shift register and select Replace with Tunnels from the pop-up menu.

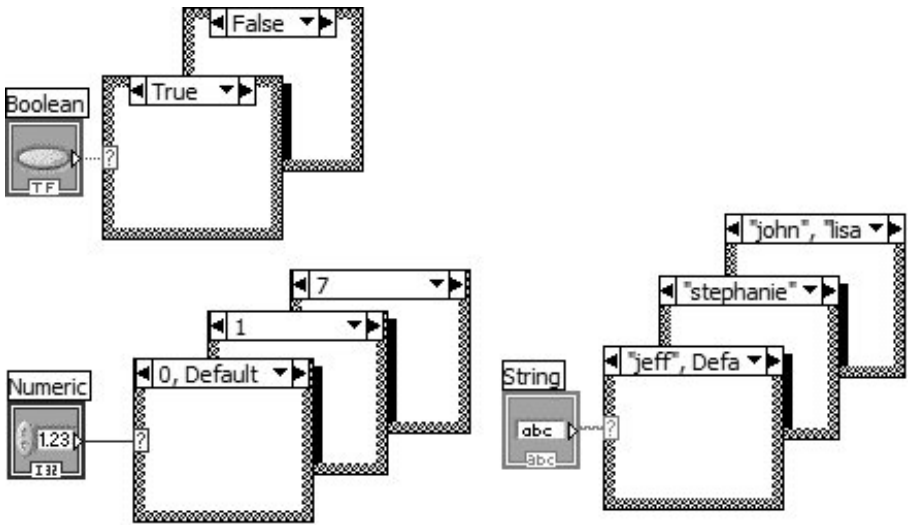
PREV NEXT

The Case Structure

Case Selector Terminal

That's enough about loops for nowlet's move on to another powerful structure. A *Case Structure* is LabVIEW's method of executing conditional text, sort of like an "if-then-else" statement. You can find it in the Programming>>Structures subpalette of the Functions palette. The Case Structure, shown in Figure 6.26, has two or more subdiagrams, or cases; only one of them executes, depending on the value of the Boolean, numeric, or string value you wire to the *selector terminal*.

Figure 6.26. Three Case Structures with their subdiagrams cascaded beneath them



If a Boolean value is wired to the selector terminal, the structure has two cases: FALSE and TRUE.

If a numeric or string data type is wired to the selector, the structure can have from one to almost unlimited cases. Initially only two cases are available, but you can easily add more. You can specify more than one value for a case, separated by commas, as shown in the previous diagram. In addition, you typically must select a "Default" case that will execute if the value wired to the selector terminal doesn't match any of the other cases; this is very handy when you can't think of every possible case but want to specify a "catch-all" case.

A Case Structure must have cases that handle all possible values that might flow into the selector terminal, or the VI will be broken (not able to run). Having a "Default" case is an easy way to achieve this, but it is not the only way. Some data types, such as enums, have a small number of possible values. Also, frames for numerics can be configured to handle a range of values; for example, "3.." handles everything from 3 to the maximum integer value of the input data type and "2..2" handles everything in the range of 2 to 2.

You can designate a case for all values in a range by using a .. notation (for example,12..9). You will also notice that LabVIEW will condense a comma delimited list of integers to this notation if there are three or more contiguous values in the list. The .. notation can be used for strings too, but the resulting logic is not very intuitive.

When you first place it on the panel, the Case Structure appears in its Boolean form; it assumes numeric values as soon as you wire a numeric data type to its selector terminal.

Decrement Arrow

Increment Arrow

Case Structures can have multiple subdiagrams, but *you can only see one case at a time*, sort of like a stacked deck of cards (unlike it appears in the previous illustration, where we cheated and took several pictures). Clicking on the decrement (left) or increment (right) arrow at the top of the structure displays the previous or next subdiagram, respectively. You can also click on the display at the top of the structure for a pull-down menu listing all cases, and then highlight the one you want to go to. Yet another way to switch cases is to pop up on the structure border and select Show Case>>.

If you wire a floating-point number to the selector, LabVIEW converts it to I32 and normal case selection is applied to the converted number.

You can position the selector terminal anywhere along the left border. You must always wire

something to the selector terminal, and when you do, the selector automatically assumes that data type. If you change the data type wired to the selector from a numeric to a Boolean, cases 0 and 1 change to FALSE and TRUE. If other cases exist (2 through n), LabVIEW does not discard them, in case the change in data type is accidental. However, you must delete these extra cases before the structure can execute.

For string data types wired to case selectors, you should always specify the case values as strings between quotes (" "). If you enter a string value without quotes, LabVIEW will add the quotes for you.

When specifying a case as the default frame, never use quotes around the keyword `Default`*. The use of quotes signifies a string whose value is "Default," not the default frame.*

For Case Structures that have a string type wired to their case selector terminal, you can select Case Insensitive Match from the pop-up menu to change the Case Structure from case-sensitive mode (the default mode) to case-insensitive mode.

Wiring Inputs and Outputs

The data at all Case Structure input terminals (tunnels and selector terminal) is available to all cases. Cases are not required to use input data or to supply output data, *but if any one case outputs a value to a tunnel, all must output a value (unless Use Default If Unwired, which we will discuss in a moment, is enabled from the pop-up menu of the output tunnel)*. When you wire an output from one case, a little white tunnel appears in the same location on all cases. The run arrow will be broken until you wire data to this output tunnel from every case, at which time the tunnel will turn black and the run arrow will be whole again (provided you have no other errors). Make sure you wire *directly* to the existing output tunnel, or you might accidentally create more tunnels.

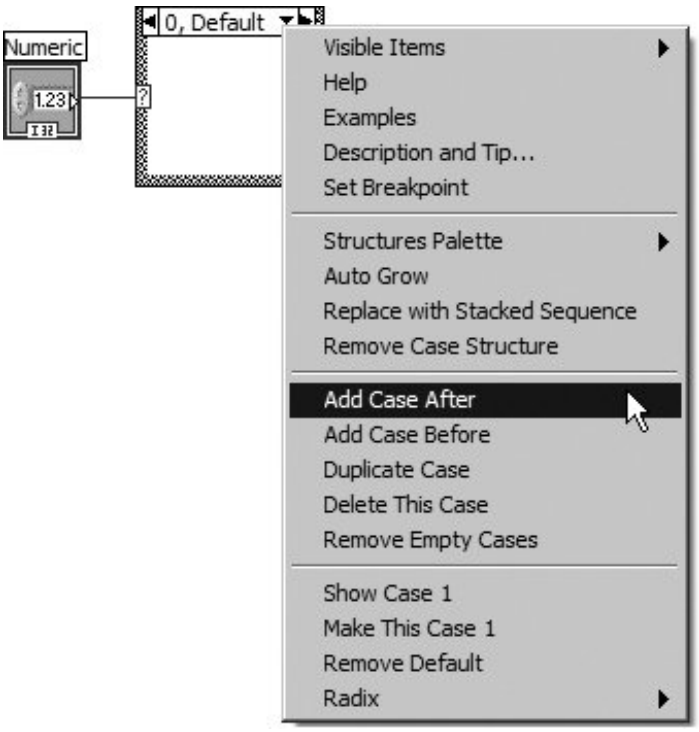
Why must you always assign outputs for each case, you ask? Because the Case Structure must supply a value to the next node regardless of which case executes. LabVIEW forces you to select the value you want rather than selecting one for you, as a matter of good programming practice.

When an output tunnel is set to Use Default If Unwired from the pop-up menu, the tunnel will output default data if the executing frame is not wired to it. This can be a useful feature but be very careful that the default data will not cause you problems. Having LabVIEW force you to wire data to the terminal in every frame of the Case Structure forces you to decide what data flows out of the terminal.

Adding Cases

If you pop up on the Case Structure border, the resulting menu gives you options to Add Case After and Add Case Before the current case (see Figure 6.27). You can also choose to copy the currently shown case by selecting Duplicate Case. You can delete the current case (and everything in it) by selecting Remove Case.

Figure 6.27. Adding a case to a Case Structure from its pop-up menu



If you would like to see some examples of how to use various structures in LabVIEW, select the Examples option from the structure's pop-up menu. This will take you to the help page, where there is a link to examples.

PREV NEXT

Dialogs

We'll switch gears for a moment; we will digress from structures to tell you about dialog boxes so that you can use them in the next activity.

Can LabVIEW carry on a conversation with you? For now, you can make LabVIEW pop-up dialog windows that have a message and often some response buttons, like "OK" and "Cancel," just like in other applications.

The dialog functions are accessible from the Programming>>Dialog & User Interface palette. LabVIEW provides you with three types of pop-up dialog boxes: one-, two-, and three-button. In each case, you can write the message that will appear in the window, and specify labels on the buttons. The two-button dialog function returns a Boolean and the three-button dialog function returns an enumerated value indicating which button was pressed. In any case, LabVIEW pauses execution of the VI until the user responds to the dialog box. The dialog box is said to be *modal*, which means that even though windows (such as other VIs' front panels) will continue to run and be updated, the user can't select any of them or do anything else via the mouse or keyboard until he or she has dealt with the dialog box.

As an example, suppose you wanted to add a dialog box to confirm a user's choice on critical selections. In the example shown in Figures 6.28 and 6.29, a dialog box is presented when the user presses the computer's self-destruct button.

Figure 6.28. Dialog.vi front panel, displaying a two-button dialog that allows the user to confirm their action

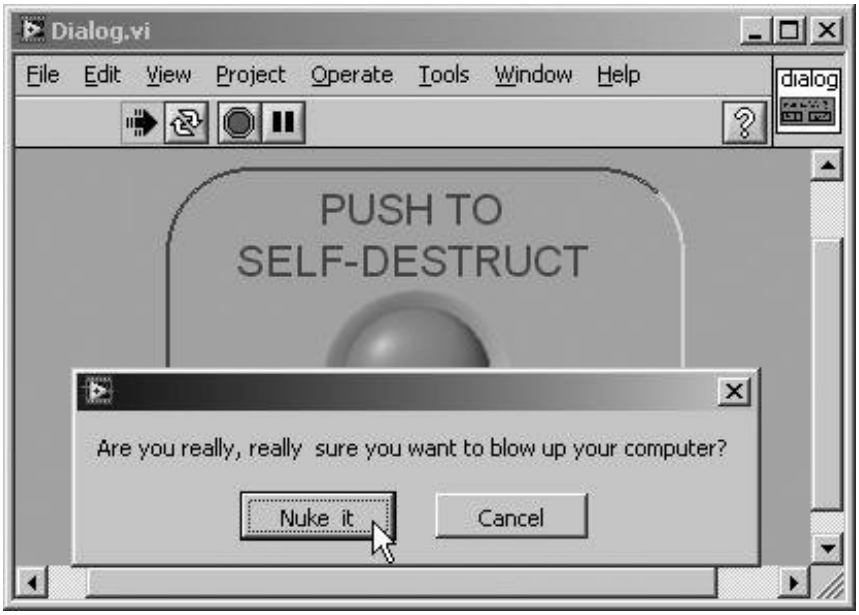
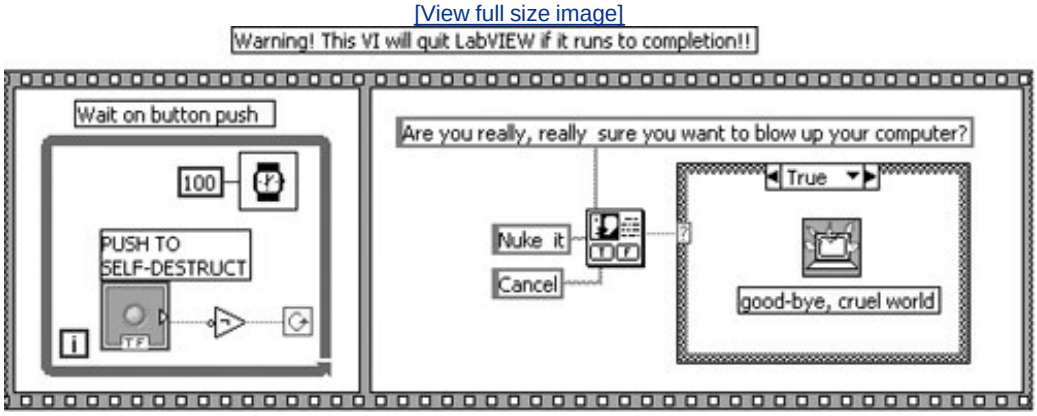


Figure 6.29. Dialog.vi block diagram, containing the Two Button Dialog function used to display the dialog

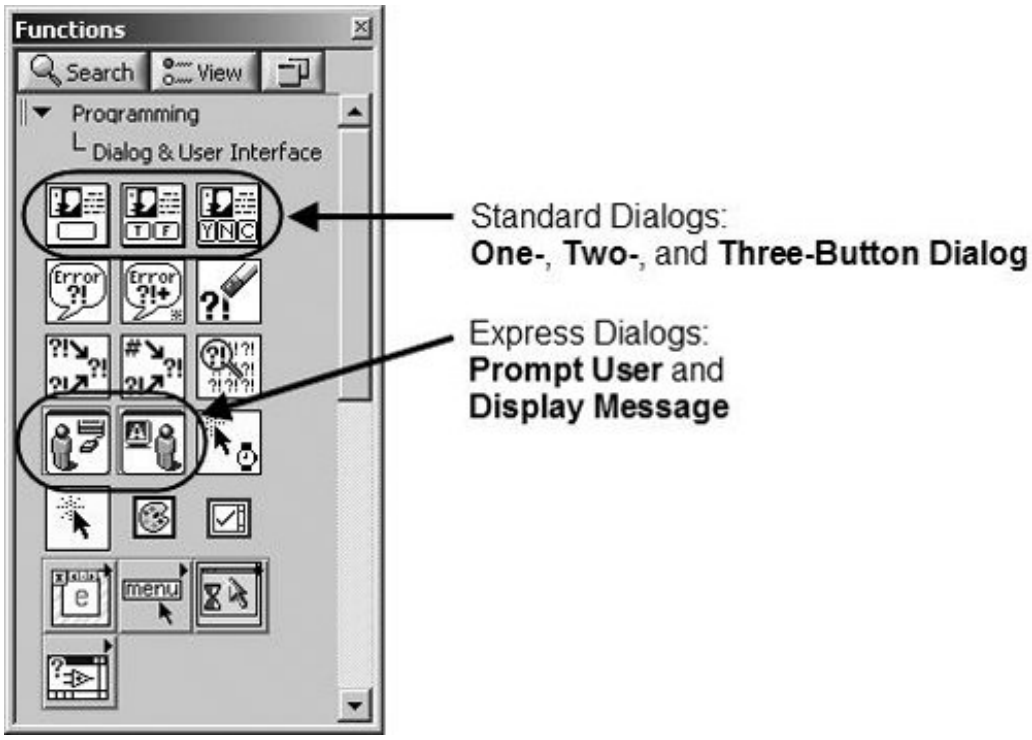
[View full size image]



The dialog functions, both standard and express, bring up a *dialog box* containing a message of your choice. You can find these functions in the Programming>>Dialog & User Interface subpalette of the Functions palette, shown in Figure 6.30.

Figure 6.30. Express and standard dialogs on the Dialog & User Interface palette

[View full size image]



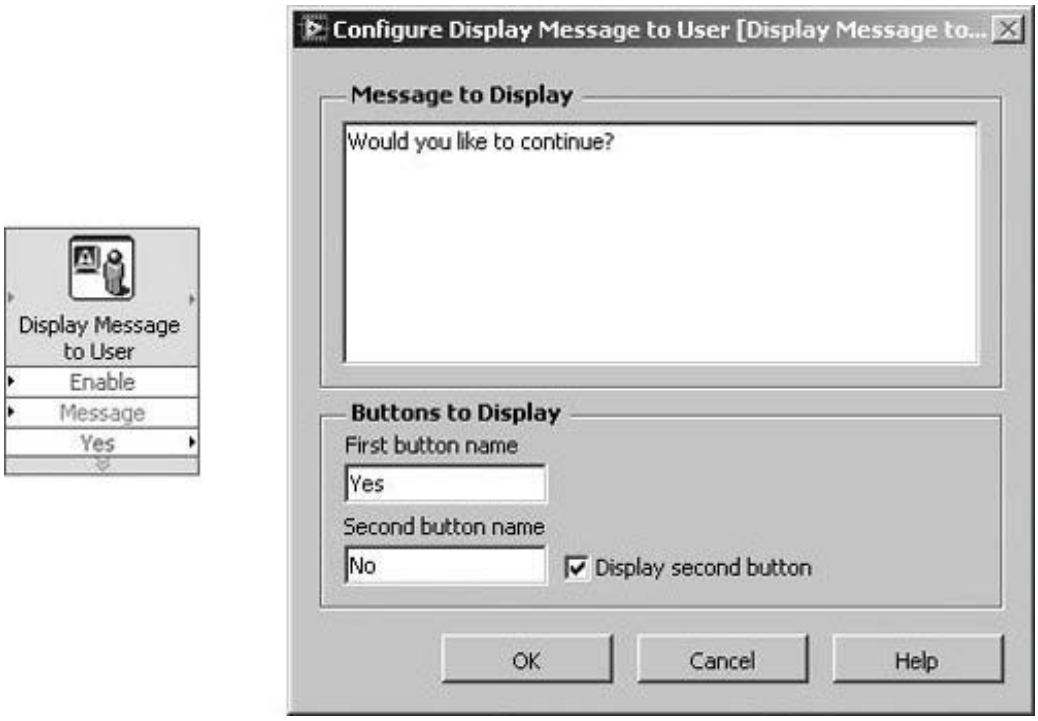
Express Dialogs: Display Message and Prompt User

The quickest way to create a dialog that displays a message or prompts the user for information is to use the Display Message and Prompt User Express VIs (respectively).

Figure 6.31 shows how easy it is to configure the Display Message express VI to ask the user a simple Yes or No question. (Open the configuration dialog by double-clicking the Express VI, or by selecting Properties from its pop-up menu.)

Figure 6.31. Display Message Express VI and its configuration dialog

[View full size image]



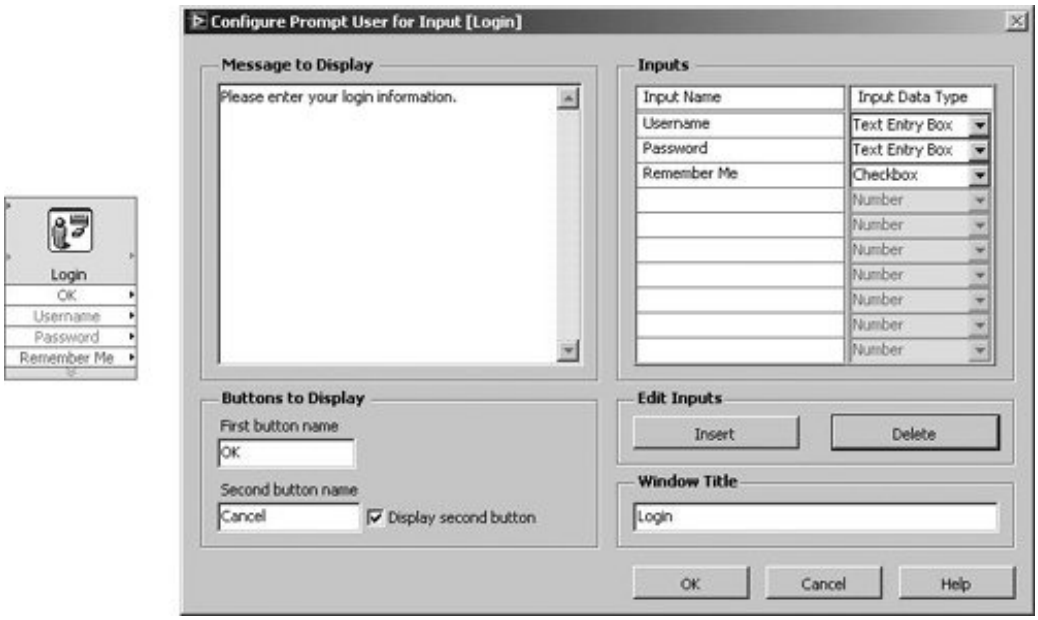
When Display Message is called, it will display a two-button dialog (because we have configured it to `Display second button`, in our example), like the one shown in Figure 6.32. (You can leave the `Display second button` checkbox unchecked to display only one button.)

Figure 6.32. Display Message Express VI dialog

For situations where you want to prompt the user to enter data, such as their name and password, you can use the Prompt User Express VI, shown in Figure 6.33.

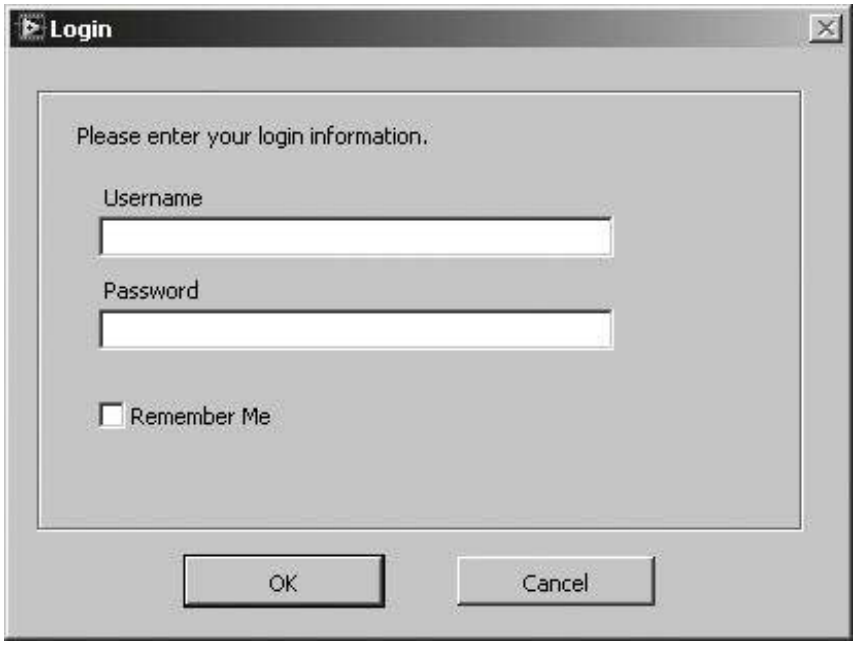
Figure 6.33. Prompt User Express VI and its configuration dialog

[View full size image]



When we call this VI, it will display a dialog (see Figure 6.34) containing the controls specified in the `Inputs` section of the configuration dialog, along with the specified text message and buttons.

Figure 6.34. Prompt User Express VI dialog



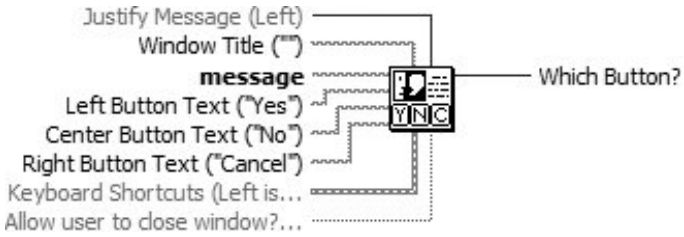
Standard Dialogs: One, Two, and Three Button Dialogs

There are three standard dialog functions: One Button Dialog, Two Button Dialog, and Three Button Dialog, shown in Figures 6.35 through 6.37.

Figure 6.35. One Button Dialog

Figure 6.36. Two Button Dialog

Figure 6.37. Three Button Dialog



The One Button Dialog stays open until you click the OK button, while the Two Button Dialog box remains until you click either the OK or the Cancel button. The Three Button Dialog remains open until you click one of the buttons or close the window, and it tells you which of these four events occurred. (The Two Button Dialog defaults to Cancel if you close the window.) You can also rename these buttons by inputting "button name" strings to the functions. These dialog boxes are *modal;* in other words, you can't activate any other LabVIEW window while they are open. They are very useful for delivering messages to or soliciting input from your program's operator.

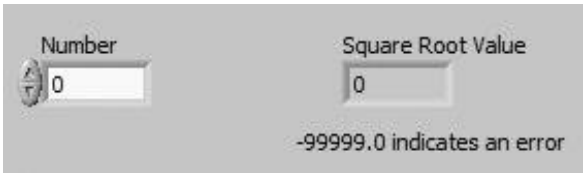
Activity 6-3: Square Roots

This activity will give you some practice with Case Structures and dialog boxes. You will build a VI that returns the square root of a positive input number. If the input number is negative, the VI pops up a dialog box and returns an error.

1. Open a new panel.

2. Build the front panel shown in Figure 6.38.

Figure 6.38. The front panel of the VI you will build during this activity



The `Number` digital control supplies the input number. The `Square Root Value` indicator will display the square root of the number.

3. Open the block diagram window. You will construct the code shown in Figures 6.39 and 6.40.

Figure 6.39. False case

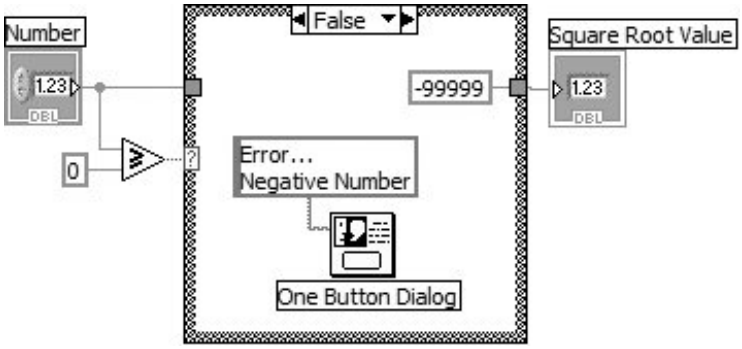
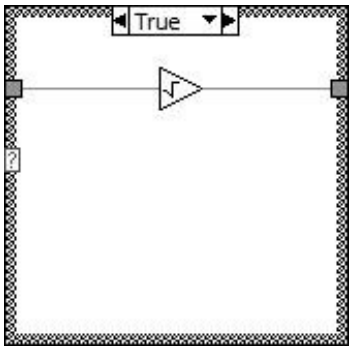


Figure 6.40. True case



4. Place the Case Structure (Programming>>Structures subpalette) in the block diagram window. As you did with the For Loop and While Loop, click with the structure cursor and drag to define the boundaries you want.

   The Greater or Equal? function returns a Boolean value, so the Case Structure remains in its default Boolean form.

   Remember, you can display only one case at a time. To change cases, click on the arrows in the top border of the Case Structure. Note that the previous picture shows two cases from the same structure so you will know what to build. *Do not create two different Case Structures for this activity!*

5. Select the other diagram objects and wire them as shown in the preceding illustration. *Make sure to use the Help window to practice displaying terminal inputs and outputs!*

Greater or Equal? Function

Greater or Equal? function (Programming>>Comparison subpalette). In this activity, checks whether the number input is negative. The function returns a TRUE if the number input is greater than or equal to zero.

Square Root Function

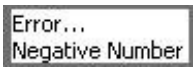
Square Root function (Programming>>Numeric subpalette). Returns the square root of the input number.

Numeric Constant

Numeric Constants (Programming>>Numeric subpalette). "99999.0" supplies the error case output, and "0" provides the basis for determining if the input number is negative.

One Button Dialog Function

One Button Dialog function (Programming>>Dialog & User Interface menu). In this exercise, displays a dialog box that contains the message "Error . . . Negative Number."



String Constant

String Constant (Programming>>String subpalette). Enter text inside the box with the Operating or Labeling tool. (You will study strings in detail in Chapter 9, "Exploring Strings and File I/O.")

In this exercise, the VI will execute either the TRUE case or the FALSE case of the Case Structure. If the input `Number` is greater than or equal to zero, the VI will execute the TRUE case, which returns the square root of the number. If `Number` is less than zero, the FALSE case outputs a 99999.00 and displays a dialog box containing the message "Error . . . Negative Number."

Remember that you must define the output tunnel for each case, which is why we bothered with the 99999.00 error case output. When you create an output tunnel in one case, tunnels appear at the same location in the other cases. Unwired tunnels look like hollow squares. Be sure to wire to the output tunnel for each unwired case, clicking on the tunnel itself each time, or you might accidentally create another tunnel.

6. Return to the front panel and run the VI. Try a number greater than zero and one less than zero.

7. Save and close the VI. Name it Square Root.vi and place it in your `MYWORK` directory.

Square Root VI Logic

```
If (Number >= 0) then
   Square Root Value = SQRT (Number)
Else
   Square Root Value = -99999.0
   Display Message "Error ... Negative Number"
End If
```

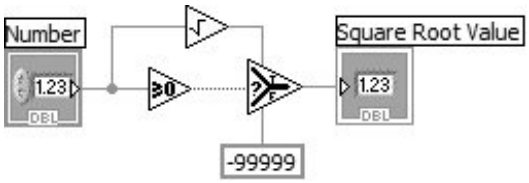
The Select Function

In simple "if-then-else" cases, you might find it more convenient to use LabVIEW's Select function, which works much like a Case Structure (see Figure 6.41).

Figure 6.41. Select function

The Select function, found in the Programming>>Comparison subpalette of the Functions palette, returns a value of t if the s input value is TRUE, and returns a value of f if the s input is FALSE. This function could accomplish almost the same thing as the Case Structure in the last activity, with the exception of popping up the dialog box (see Figure 6.42).

Figure 6.42. The Select function used to conditionally switch between two possible outputs



PREV NEXT

The Sequence StructureFlat or Stacked

Determining the execution order of a program by arranging its elements in a certain sequence is called *control flow*. Visual Basic, C, and most other procedural programming languages have inherent control flow because statements execute in the order in which they appear in the program. LabVIEW uses the *Sequence Structure* to obtain control flow within a dataflow framework. A Sequence Structure is an ordered set of frames that execute sequentially. A Sequence Structure executes frame 0, followed by frame 1, then frame 2, until the last frame executes. Only when the last frame completes does data leave the structure.

There are two flavors of Sequence Structure: the *Flat Sequence Structure* and the *Stacked Sequence Structure*, shown in Figure 6.43 and Figure 6.44, respectively. These can both be found in the Programming>>Structures subpalette of the Functions palette. The two types of Sequence Structure are almost identical, having frames that look like frames of a film. The difference is that the frames of a Flat Sequence Structure are organized sequentially side-by-side and the frames of a Stacked Sequence Structure are organized sequentially in a stack, appearing similar to a Case Structure.

Figure 6.43. Flat Sequence Structure

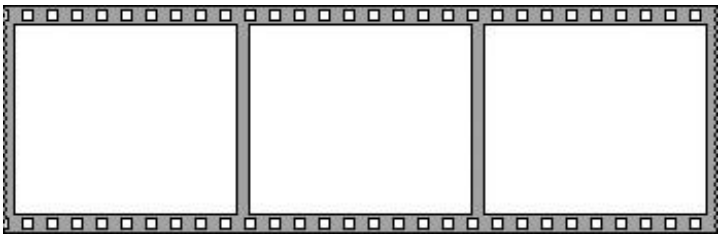
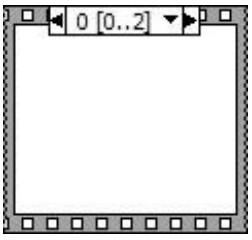


Figure 6.44. Stacked Sequence Structure



Regardless of appearance, either type of Sequence Structure (Flat or Stacked) executes code exactly

in the same manner. In fact, you can easily convert one into the other by selecting from its pop-up menu, Replace with Stacked Sequence (to convert from Flat to Stacked) or Replace>>Replace with Flat Sequence (to convert from Stacked to Flat).

Don't let the size advantage of the Stacked Sequence Structure lure you into thinking that it is a better choice than the Flat Sequence Structure. As we will soon find out, the use of Stacked Sequence Structures can lead to many negative side effects.

Stacked Sequence Structures and Sequence Locals Are Evil

OK, the title of this section is pretty harsh. Sequence Locals aren't really evilthey have no minds of their own. However, we want to be sure that you know to avoid using them or you may find yourself heading down the dark and dangerous path toward a VI full of spaghetti code. Let's take a look at the Stacked Sequence Structure and Sequence Locals, and then we will look at an example that shows why we should avoid them.

Like the Case Structure, the Stacked Sequence Structure has only one frame visible at a timeyou must click the arrows at the top of the structure to see other frames, or you can click on the top display for a listing of existing frames, or pop up on the structure border and choose Show Frame . . . . When you first drop a Sequence Structure on the block diagram, it has only one frame; thus, it has no arrows or numbers at the top of the structure to designate which frame is showing. Create new frames by popping up on the structure border and selecting Add Frame After or Add Frame Before.

There are some very important reasons why the Flat Sequence Structure is a better choice than the Stacked Sequence Structure:

- Because all the frames of a Flat Sequence Structure are visible, you can see all of your code at once.

Figure 6.45. Sequence Local cannot be used.

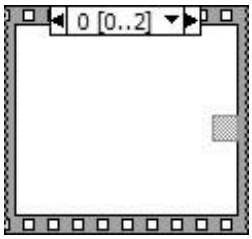


Figure 6.46. Sequence Local is a data sink, in the frame that initializes its value.

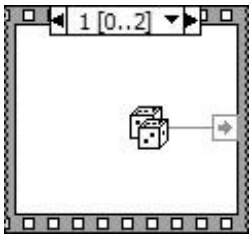


Figure 6.47. Sequence Local is a data source, after it has been initialized; it can be wired to an indicator (a data sink).

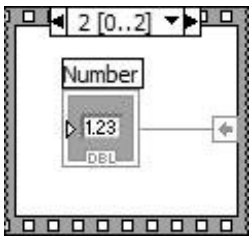
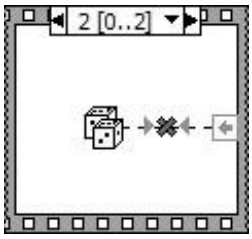


Figure 6.48. Sequence Local is a data source, after it has been initialized; you cannot wire an input value (another source) to it.



- Data is passed between frames of a Flat Sequence Structure using tunnels, rather than Sequence Locals (which are evil).

Sequence Locals are evil because

- They force you to break the left-to-right convention of data-flow programming.

- You cannot easily see which frame initializes a Sequence Local.

- They tend to multiplyyou will generally need more Sequence Locals in a Stacked Sequence Structure than tunnels in a Flat Sequence Structure with the same frames of code.

Figure 6.49 and Figure 6.50 show the dramatic difference between the Flat Sequence Structure and the Stacked Sequence Structure. It is easy to see the difference in readability.

Figure 6.49. A simple example using a Flat Sequence Structureclean!

[View full size image]

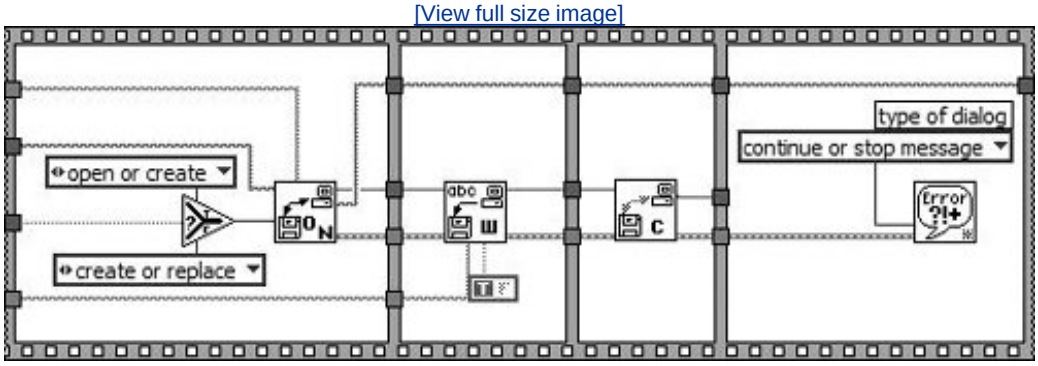
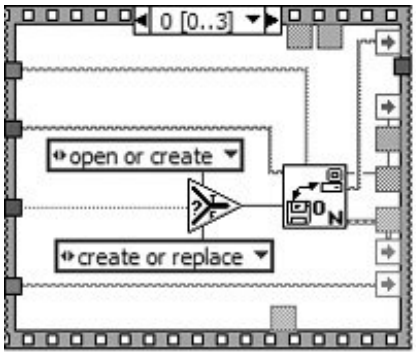
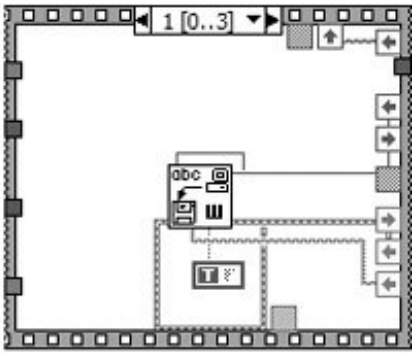
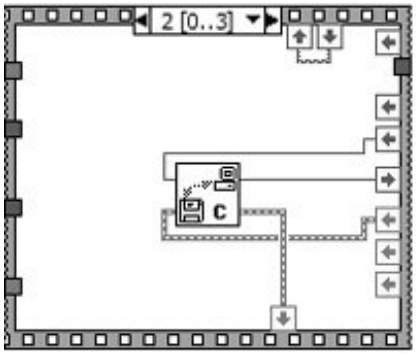


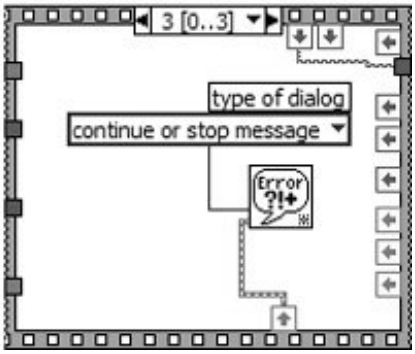
Figure 6.50. A Stacked Sequence equivalent of our Flat Sequence Structure exampleyuck!

[View full size image]


0 [0..3]
open or create
create or replace


1 [0..3]
ODC


2 [0..3]


3 [0..3]
type of dialog
continue or stop message
Error

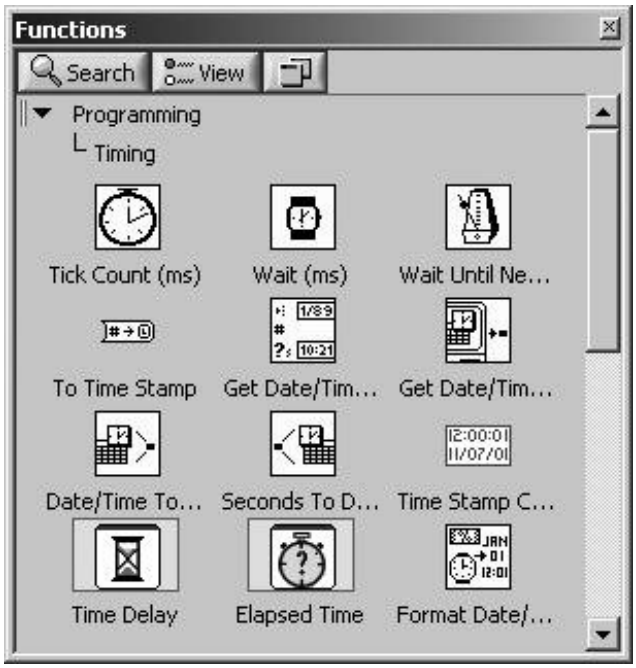
PREV NEXT

Timing

It's "time" for another digression. Timing functions are very important in LabVIEW and help you measure time, synchronize tasks, and allow enough idle processor time so that loops in your VI don't race too fast and hog the CPU.

You can find LabVIEW's timing functions on the Programming>>Timing palette, as shown in Figure 6.51.

Figure 6.51. Timing palette



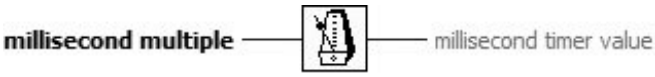
The basic timing functions in LabVIEW are Wait (ms), Tick Count (ms), and Wait Until Next ms Multiple, located in the Programming>>Timing subpalette of the Functions palette.

Wait (ms) causes your VI to wait a specified number of milliseconds before it continues execution (see Figure 6.52).

Figure 6.52. Wait (ms)

Wait Until Next ms Multiple causes LabVIEW to wait until the internal clock equals or has passed a multiple of the millisecond multiple input number before continuing VI execution; it is useful for causing loops to execute at specified intervals and synchronizing activities (see Figure 6.53). These two functions are similar, but not identical. For example, Wait Until Next ms Multiple will probably wait less than the specified number of milliseconds in the first loop iteration, depending on the value of the clock when it starts (that is, how long it takes until the clock is at the next multiple and the VI proceeds).

Figure 6.53. Wait Until Next ms Multiple



Wait Until Next ms Multiple is commonly used for synchronizing one or more loops because its timing is periodicit waits until a specific time occurs. Wait (ms) is commonly used to create a pause between eventsit waits for a specified period of time to elapse.

Tick Count (ms) returns the value of your operating system's internal clock in milliseconds; it is commonly used to calculate elapsed time, as in the next activity (see Figure 6.54).

Figure 6.54. Tick Count (ms)

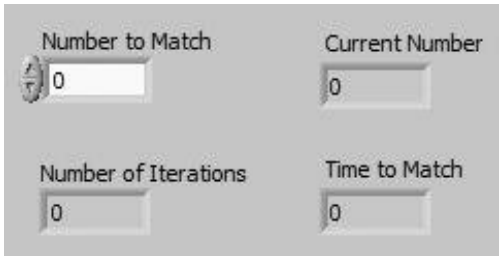
Activity 6-4: Matching Numbers

Now you'll have the opportunity to work with the Sequence Structure and one of the timing functions. You will build a VI that computes the time it takes to match an input number with a randomly generated number. Remember this algorithm if you ever need to time a LabVIEW operation. Many experienced LabVIEW developers believe that this is the only reason ever to use a multiframe Sequence Structurefor enforcing and measuring execution timing.

1. Open a new front panel.

2. Build the front panel shown (see Figure 6.55).

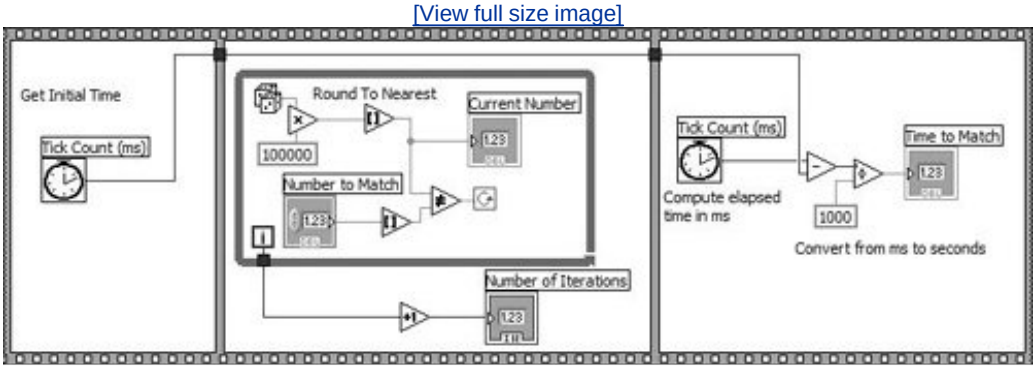
Figure 6.55. The front panel of the VI you will construct during this activity



3. Open the diagram window and build the block diagram shown in Figure 6.56.

Figure 6.56. The block diagram of the VI you will construct during this activity

[View full size image]



4. Place a Flat Sequence Structure (Programming>>Structures palette) in the diagram window. It works like the For Loop and While Loop; click with the structure cursor and drag to define the boundaries you want.

   You will have to build three separate frames of the Flat Sequence Structure. To create new frames, pop up on the frame border and choose Add Frame After or Add Frame Before from the pop-up menu.

5. Build the rest of the diagram. Some new functions are described here. Make sure to use the Help window to display terminal inputs and outputs when you wire!

Tick Count Function

Tick Count (ms) function (Programming>>Timing palette). Returns the value of the internal clock.

Random Number Function

Random Number (0-1) function (Programming>>Numeric palette). Returns a random number between 0 and 1.

Multiply Function

Multiply function (Programming>>Numeric palette). Multiplies the random number by 100 so that the function returns a random number between 0.0 and 100.0.

Round to Nearest Function

Round to Nearest function (Programming>>Comparison palette). Rounds the random number between 0 and 100 to the nearest whole number.

Not Equal? Function

Not Equal? function (Programming>>Comparison palette). 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 a FALSE.

Increment Function

Increment function (Programming>>Numeric palette). Adds one to the loop count to produce the `Number of Iterations` value (to compensate for zero-based indexing).

In the first frame (on the left), the Tick Count (ms) function returns the value of the internal clock in milliseconds This value is wired through to the last frame. In the second frame (in the middle), the VI executes the While Loop as long as the number specified does not match the number returned by the Random Number (0-1) function. In the final frame (on the right), the Tick Count (ms) function returns a new time in milliseconds. The VI subtracts the old time from the new time to compute the time elapsed, and then divides by 1000 to convert from milliseconds to seconds.

6. Turn on execution highlighting, which slows the VI enough to see the current generated number on the front panel.

7. Enter a number inside the `Number to Match` control and run the VI. When you want to speed things up, turn off execution highlighting.

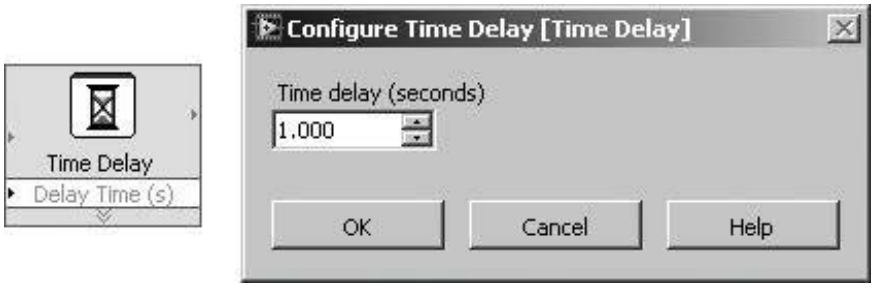
8. Use the Save command to save the VI in your `MYWORK` directory as Time to Match.vi, and then close it. Good job!

Express Timing Functions

In addition to the basic timing functions, LabVIEW also provides you with two Express Timing functions: Time Delay and Elapsed Time.

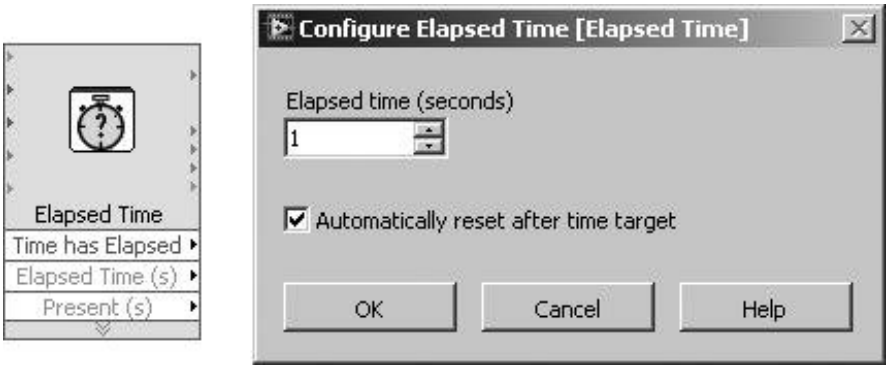
Time Delay works just like Wait (ms) except that you specify the time delay in seconds (see Figure 6.57).

Figure 6.57. Time Delay Express VI and its configuration dialog



Elapsed Time lets you check whether a specified amount of time has passed. When you configure this timing function, you set how many seconds of elapsed time you want it to check for. When you call the VI, the Boolean output "Time has Elapsed" will return a TRUE if the specified amount of time has elapsed; otherwise, it returns a FALSE (see Figure 6.58).

Figure 6.58. Elapsed Time Express VI and its configuration dialog

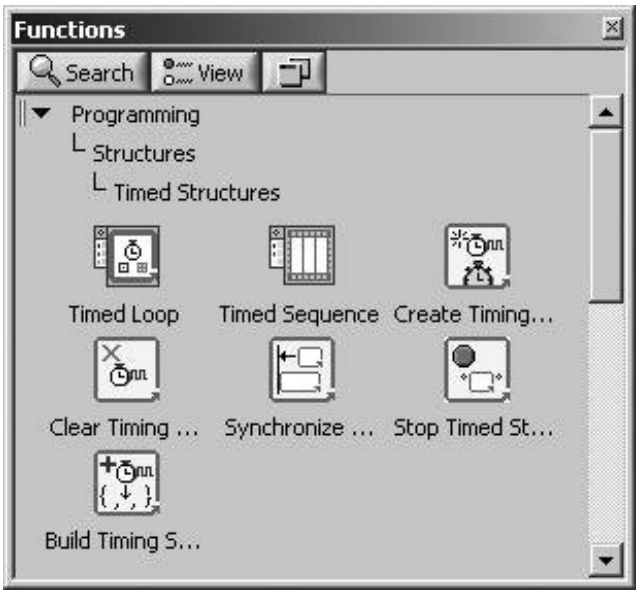


PREV NEXT

The Timed Structures

Doesn't it seem like every While Loop has some kind of timing function inside of it? Wouldn't it be great if the timer were somehow built into the While Loop? Well, if you take a look at the Programming>>Structures>>Timed Structures palette (shown in Figure 6.59), you'll find a set of tools that make While Loop (as well as Sequence Structure) timing and synchronization possible.

Figure 6.59. Timed Structures palette



Timed Structures are only available on Windows. They were designed especially for use in time-critical LabVIEW RT and FPGA applications, and those LabVIEW modules are not supported on Mac OS X and Linux.

Timed Structures and VIs allow you control the rate and priority at which a timed structure executes its subdiagram, synchronize the start time of timed structures, create timing sources, and establish a hierarchy of timing sources.

The Timed Loop and the Timed Sequence (which you'll learn about next) have several nodes attached to their frames. These are used for configuring the loops and obtaining information about their execution.

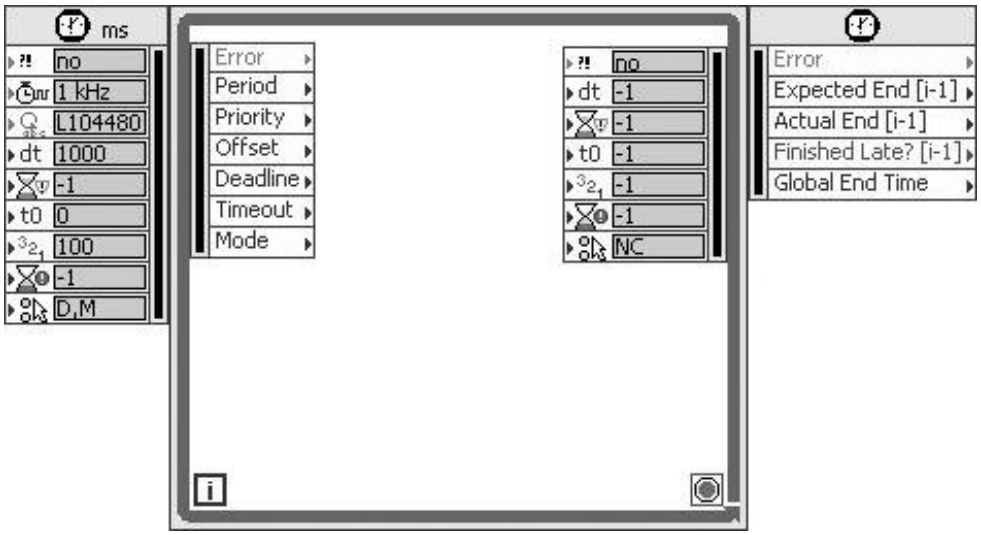
Probably the biggest conceptual hurdle in using the timed structures is that each timed structure may be named *(using a string). The name string is passed into the Name terminal on the outer-left node of the Timed Structure. This name is then used as a reference to the timed structure when operating on it using the time structure VIs, which you will also learn about shortly. Similarly,* timing sources *and* synchronization groups *are also* named *(using a string) and referenced by their names.*

OK, now you're ready for the nitty gritty details of the timed structures and VIs.

The Timed Loop

The Timed Loop (shown in Figure 6.60) executes one or more subdiagrams, or frames, sequentially each iteration of the loop at the period you specify. Use the Timed Loop when you want to develop VIs with multi-rate timing capabilities, precise timing, feedback on loop execution, timing characteristics that change dynamically, or several levels of execution priority. Right-click the structure border to add, delete, insert, and merge frames. If you use the Timed Loop in an FPGA VI, the loop executes one subdiagram at the same period as an FPGA clock.

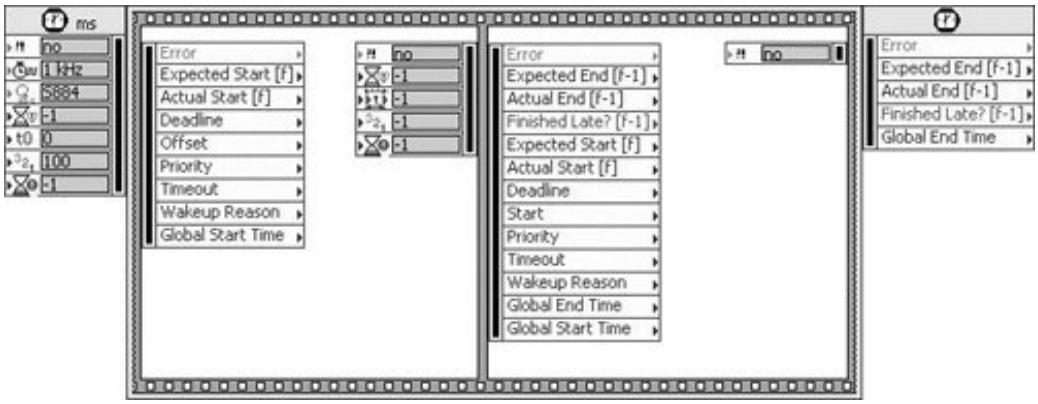
Figure 6.60. The Timed Loop structure



The Timed Sequence

The Timed Sequence (shown in Figure 6.61) consists of one or more task subdiagrams, or frames, that execute sequentially. Use the Timed Sequence when you want to develop VIs with multi-rate timing capabilities, precise timing, execution feedback, timing characteristics that change dynamically, or several levels of execution priority. Right-click the structure border to add, delete, insert, and merge frames.

Figure 6.61. The Timed Sequence structure



The Timed Structure VIs

The following VIs are used to control timed structures and their timing sources (see Figures 6.626.66).

Figure 6.62. Build Timing Source Hierarchy

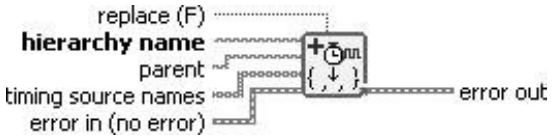


Figure 6.63. Clear Timing Source

Figure 6.64. Create Timing Source

Figure 6.65. Stop Timed Structure

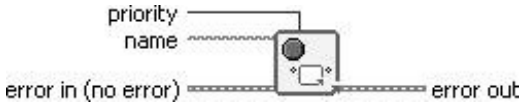
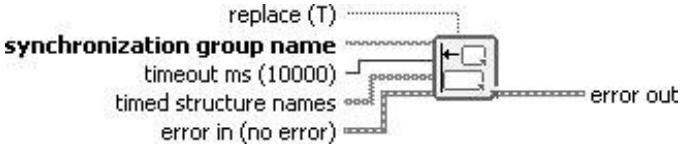


Figure 6.66. Synchronize Timed Structure Starts



Creates a hierarchy of timing sources based on the names you enter in timing source names. The hierarchy determines the order in which the timing sources start. The parent timing source does not start until after the timing sources in hierarchy name start. Use this VI when you have timing

sources that have signal dependencies, such as DAQ counters routed to drive analog input connections. In such a case, the counter timing source acts as the parent.

Stops and deletes the timing source you created or specified for use by another resource. If the timing source is associated with a DAQmx task, the VI also clears the task. The VI cannot reuse a name until all Timed Loops attached to the timing source terminate.

Creates a timing source you use as the timing source in the Timed Loop. Each timing source has its own unit of timing and/or start time and does not start until the first Timed Loop that uses the timing source starts. You must manually select the polymorphic instance you want to use. (You can choose from either a 1kHz timing source or a 1 MHz timing source.)

Stops the Timed Loop or Timed Sequence you enter in name. If you attempt to abort a running Timed Loop, the Timed Loop executes another iteration and returns Aborted in the Wakeup Reason output of the Left Data node.

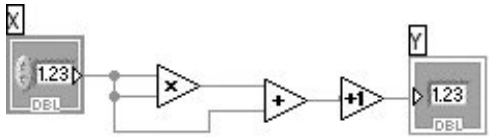
Synchronizes the start of Timed Loops or Timed Sequences you enter in timed structure names by adding the names to the synchronization group you specify in synchronization group name. All timed structures in a synchronization group wait until all the structures are ready to execute.

PREV NEXT

The Formula Node

Now that you know about LabVIEW's four main control flow structures, we'll introduce a structure that doesn't affect program flow. The *Formula Node* is a resizable box that you use to enter algebraic formulas directly into the block diagram. You will find this feature extremely useful when you have a long formula to solve. For example, consider the fairly simple equation, $y = x^2 + x + 1$. Even for this simple formula, if you implement this equation using regular LabVIEW arithmetic functions, the block diagram is a little bit harder to follow than the text equations (see Figure 6.67).

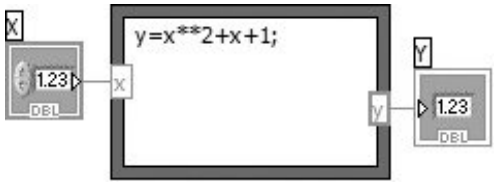
Figure 6.67. A snippet of code that we can convert to a formula and put inside a Formula Node, as shown in Figure 6.68



You can implement the same equation using a Formula Node, as shown in Figure 6.68.[1]

[1] In versions prior to LabVIEW 6.0, the exponentiation operator for the Formula Node was the ^ symbol. In LabVIEW 6.0 and greater, the ^ symbol means something else entirely (bit-wise XOR), and the ** symbol is exponentiation.

Figure 6.68. A Formula Node containing a formula derived from the code snippet in Figure 6.67



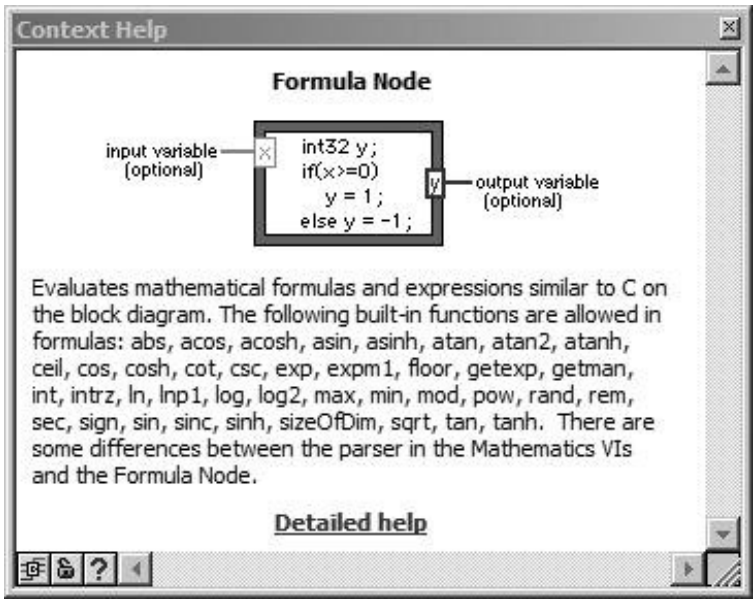
With the Formula Node, you can directly enter a formula or formulas, in lieu of creating complex block diagram subsections. Simply enter the formula inside the box. You create the input and output terminals of the Formula Node by popping up on the border of the node and choosing Add Input or Add Output from the pop-up menu. Then enter variable names into the input and output terminals. Names are case sensitive, and *each formula statement must terminate with a semicolon (;)*.

You will find the Formula Node in the Programming>>Structures subpalette of the Functions palette.

These operators and functions are available inside the Formula Node.

To get detailed information on the Formula Node syntax, open the Help window and place the cursor over the formula node. The Help window will look like Figure 6.69. Click on the Detailed help link to open the LabVIEW help file and then follow the Formula Node Syntax link.

Figure 6.69. The Context Help window showing detailed information about the Formula Node

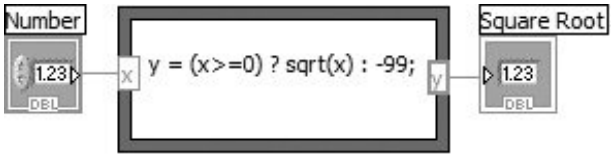


The following example shows a conditional branching that you could perform inside a Formula Node. Consider the following code fragment, similar to Activity 6-3, which computes the square root of x if x is positive, and assigns the result to y. If x is negative, the code assigns value of 99 to y.

```
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 Figure 6.70.

Figure 6.70. A Formula Node formula with some of its syntax elements annotated

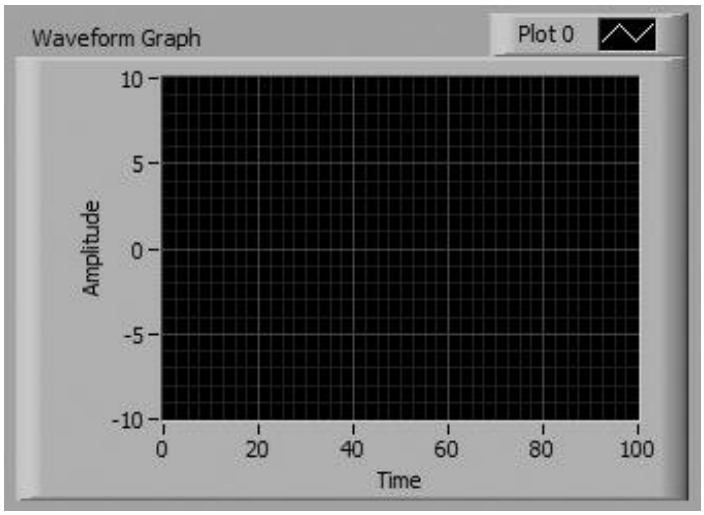


Activity 6-5: Formula Fun

You will build a VI that uses the Formula Node to evaluate the equation `y = sin(x)`, and graph the results.

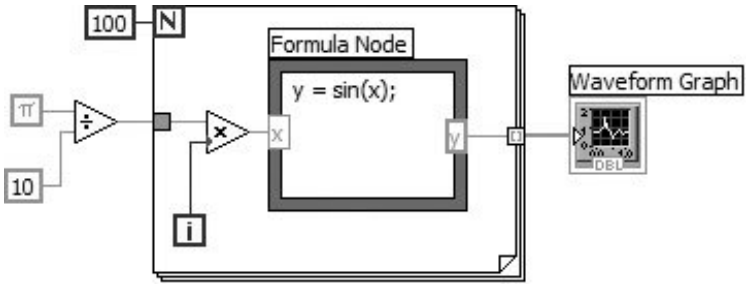
1. Open a new panel. Select Waveform Graph from the Modern>>Graph subpalette of the Controls palette (see Figure 6.71). Label it Graph. You'll learn all about graphs in Chapter 8, but this activity would be kind of dull without a nice pictorial representation, so we thought we'd give you a taste of them.

Figure 6.71. The front panel of the VI you will build during this activity



2. Build the block diagram shown in Figure 6.72.

Figure 6.72. The block diagram of the VI you will build during this activity



With the Formula Node (Structures palette), you can directly enter mathematical formulas. Create the input terminal by popping up on the border and choosing Add Input from the pop-up menu; then 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. Remember, variable names are case sensitive.

Notice that a semicolon (;) must terminate each statement in the formula node.


Pi Constant

The *p* constant is located in the Functions>>Programming>>Numeric>>Math and Scientific Constants palette.

During each iteration, the VI multiplies the iteration terminal value by p/10. The multiplication result is wired to the Formula Node, which computes the sine of the result. The VI then stores the result in an array at the For Loop border. (You will learn all about arrays in Chapter 7. Then you will see why you can wire array data out of a For Loop, while scalar data comes out of a While Loop by default.) After the For Loop finishes executing, the VI plots the array.

3. Return to the front panel and run the VI. Note that you could also use the existing Sine function (Functions>>Numeric>>Trigonometric palette) to do the same thing as the Formula Node in this activity, but LabVIEW does not have built-in functions for every formula you'll need, and we wanted to give you the practice.

4. Save the VI in your MYWORK directory and name it Formula Node Exercise.vi. Close the VI.

VI Logic

```
timebase = PI/10
for i = 0 to 99
      x = i * timebase
      y = sin(x)
      array[i] = y
next i
Graph (array)
```

PREV NEXT

The Expression Node

The Expression Node is found on the Programming>>Numeric palette (see Figure 6.73) and is basically just a simplified Formula Node having just one unnamed input and one unnamed output.

Figure 6.73. Expression Node shown on the block diagram (left) and on the Programming>>Numeric palette (right)

[View full size image]

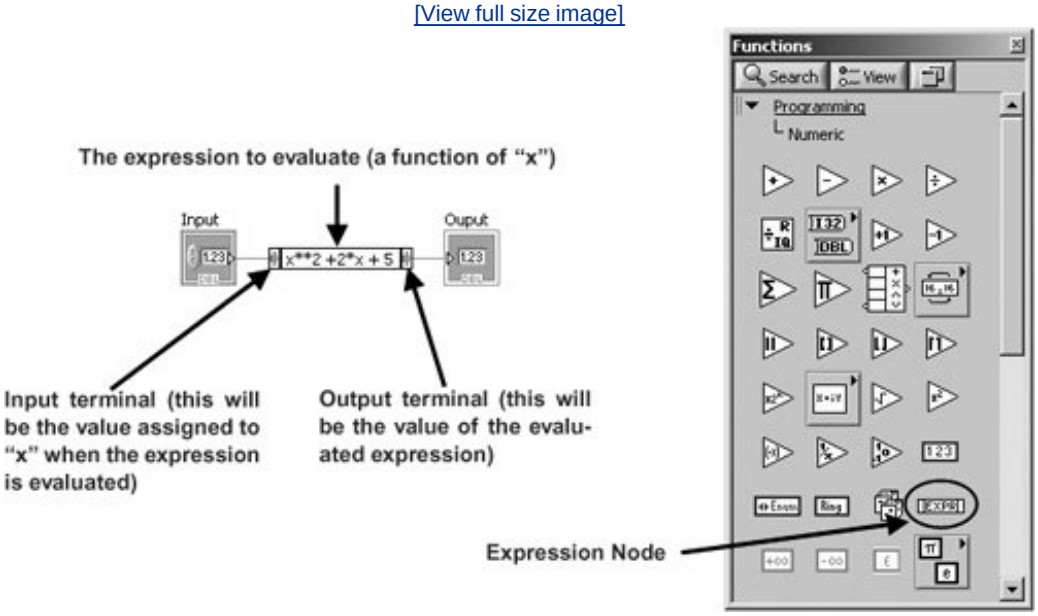


Figure 6.73 shows the Expression Node with annotated parts. Unlike the Formula Node, you do not have to name the input or output terminals. (Because there is only one input and one output, there is no ambiguity.) Also, because there is only one expression, there is no need for a semicolon at the end of the expression. (The Formula Node does require semicolons at the end of each expression.) Finally, in an Expression Node, you can use any valid name you want for the input variable (in our previous example, "x" could have just as easily been "y" or "Tps" with the same effect), but remember you can only use *one* variable in an Expression Node, unlike the Formula Node, which allows multiple input and output variables.

The same operators and syntax of the Formula Node apply to the Expression Node.

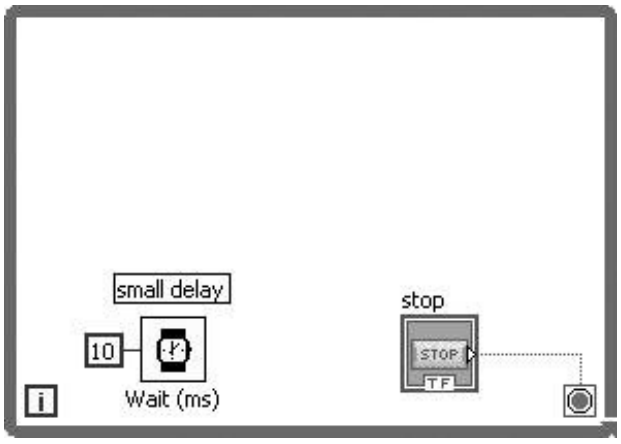
The While Loop + Case Structure Combination

Like most tools, you need a combination of them to build something non-trivial. You can't really build anything with just a hammer or just a saw. But, if you combine the capabilities of these two, you can do some amazing things. Throw in a tape measure and you're unstoppable! LabVIEW structures (our tools, in this case) are the same way. We rarely use them independently. We start with a single While Loop, our high-powered circular saw that keeps spinning until the job is done. Then we drop in a Case Structure (or Event Structure, as we will learn in Chapter 13, "Advanced LabVIEW Structures and Functions"), our high-powered pneumatic nail gun, for nailing down all the cases that our software must handle. Analogies aside, the While Loop + Case Structure combination is a very powerful combination, from which you can build upon to create just about any software application that your mind can imagine.

The Main Loop

Nearly all applications written in LabVIEW will have at least one *main* loop that will keep looping until the program terminates, due to a stop button press, error, or other exit condition. Figure 6.74 shows an example of a While Loop, along with a standard wait timer (to avoid hogging the CPU) and a stop button.

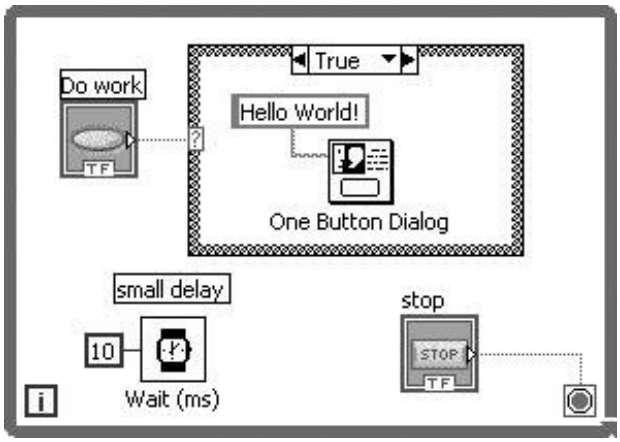
Figure 6.74. Wait (ms) timer used in a While Loop to avoid hogging the CPU



To this loop, we add a Case Structure and connect a button called `Do work`. In the TRUE case of the Case Structure, we place our workfunctions or subVIs that do something useful. Figure 6.75 shows an example of this, which opens a "Hello World" dialog each time the Do work button is pressed.

(Note that `Do work` is configured for Mechanical Action>>Latch When Released, so that it will rebound to FALSE after its terminal is read on the block diagram. This ensures that the dialog only appears once, for every button push.)

Figure 6.75. Case Structure inside a While Loop, used to execute work when the user presses a button

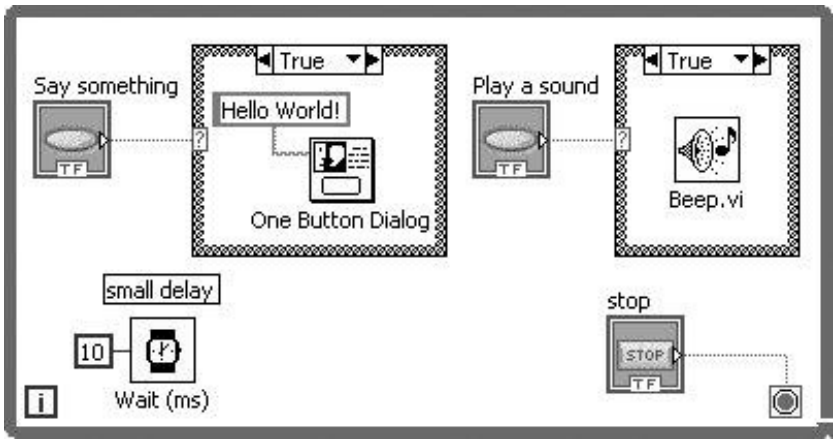


Spend a minute and let this concept really sink in. This While Loop + Case Structure combination is the key to building software applications in LabVIEW.

Handling Multiple Work Items in a While Loop

OK, you're ready for the next step. How do you handle multiple buttons? The easy way is to just add more Case Structures (well, it starts out easy, but gets tricky as you add more and more Case Structures), as shown in Figure 6.76.

Figure 6.76. Two "work handling" Case Structures inside a While Loop



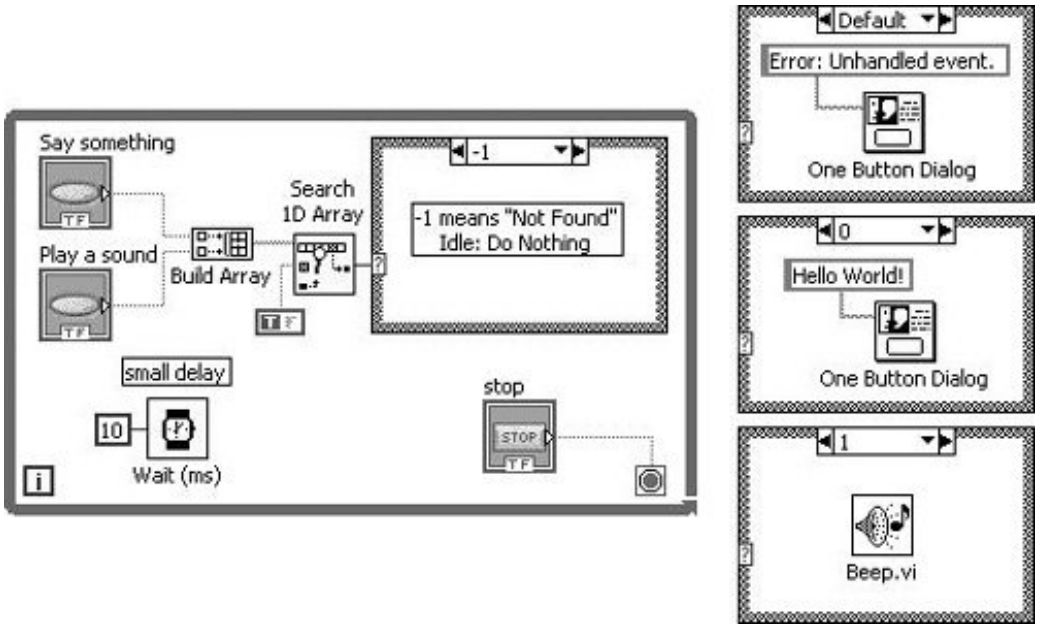
Now we know how to handle multiple work items in our While Loop + Case Structure combinationhandling *N* work items means that we will need *N* Case Structures. But what happens when we run out of room on our block diagram? It is easy to see that this "just add another Case Structure" pattern does not scale up past a handful of work items (although we have seen some block diagrams where people have tried to see how far this pattern scales and, believe us, it's not pretty).

So, how do we solve this scalability problem? The answer is simple. *We need to use a single Case Structure with multiple cases: one case for handling each work item.*

In order to achieve this, we will need to build a Boolean array of button values and search for the index of the element that is TRUE, as shown in Figure 6.77. (You will learn about arrays in Chapter 7.) In simple terms, we will figure out which button number was pressed and wire this button number into the case selector terminal of the Case Structure.

Figure 6.77. One Case Structure with multiple cases handling multiple tasks inside a While Loop

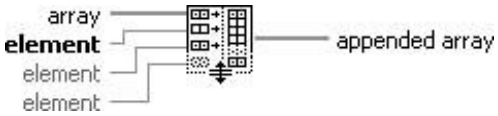
[View full size image]



Don't feel the need to understand the array manipulation happening in Figure 6.77. This will make a lot more sense after you learn about arrays in Chapter 7.

In this example, Build Array (found on the Programming>>Array palette) converts two Booleans into a 1D array of Booleans (see Figure 6.78).

Figure 6.78. Build Array



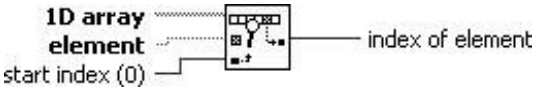
In this example, Search 1D Array (found on the Programming>>Array palette) tells us which button number was pressed (see Figure 6.79):

1: No button was pressed

0: Say something was pressed

1: Play a sound was pressed

Figure 6.79. Search 1D Array



Now, if we ever want to expand our code, we simply add another button, expand the build array node (add another terminal to the bottom) and wire our button to it, and then add another case for handling the button press.

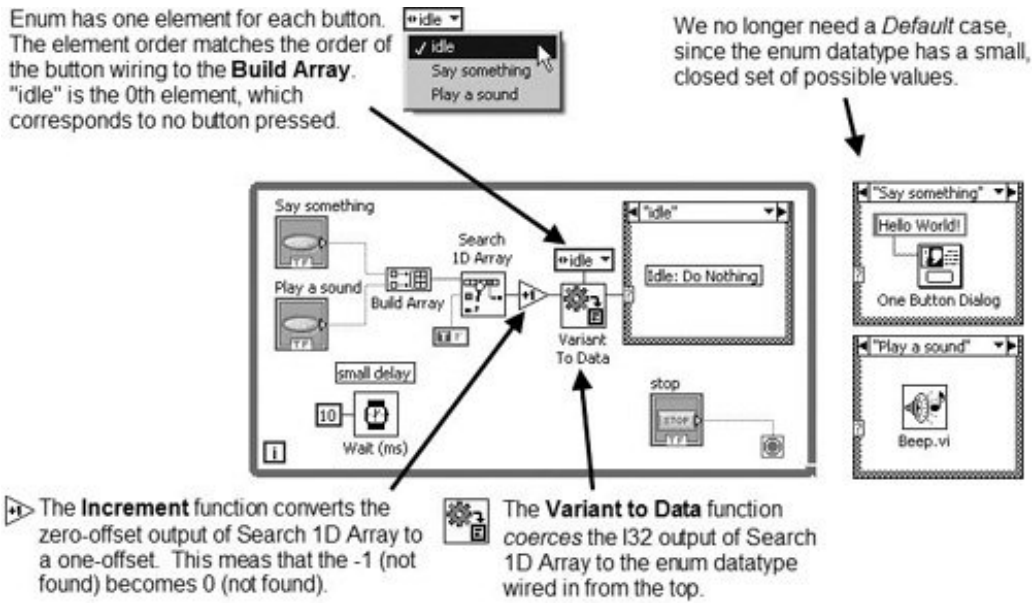
Incidentally, you can run this example if you like. It's in the `EVERYONE\CH06` directory on the CD, called While Loop with Case Structure Example.vi.

Note that we put an error message dialog in the Default frame of the Case Structure. This will alert us if we should ever add another button, but forget to add a case to handle it.

It is a common practice to use an enum datatype, wired to the case selector terminal, instead of a "plain" I32 numeric. The example is shown in Figure 6.80.

Figure 6.80. "Enumerating" Case Structure cases for handling multiple work items in a While Loop

[View full size image]



Note that this is using some advanced functions and techniques. Don't feel the need to understand everything shown here. Just make a mental note and then come back once you are ready to review it again.

When adding more work items to a While Loop + Case Structure combination that uses an enum type (as shown in Figure 6.80), you will need to remember to add additional elements to the enum constant before you can add the additional case to the Case Structure.

Adding Efficiency: Wait on Front Panel Activity

Wait For Front Panel Activity function

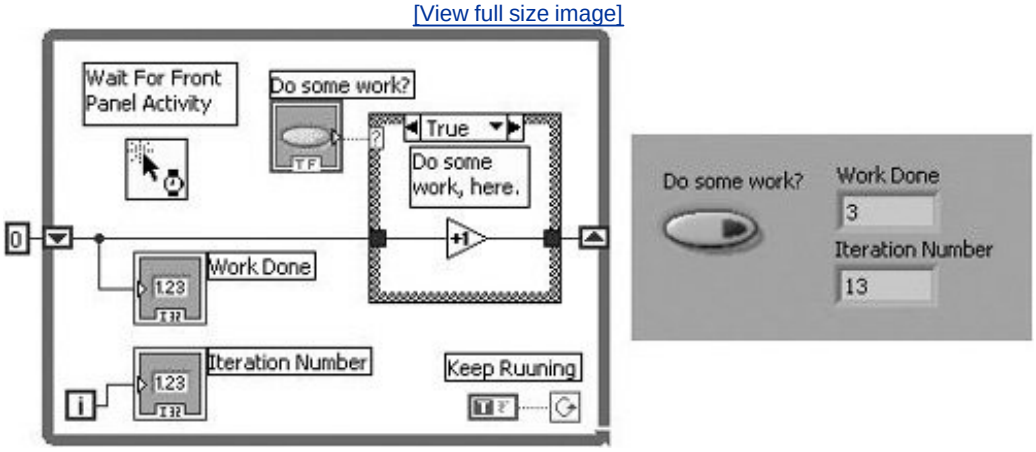
Let's take a slight detour away from the Event Structure, for just one moment, to discuss a very simple option for improving the efficiency of a While Loop that polls front panel controls.

We can use the Wait For Front Panel Activity function (found on the Programming>>Dialog & User Interface palette) to wait until front panel activity is detected, instead of using the Wait Until Next ms Multiple function to poll the user interface at a periodic interval.

Figure 6.81 shows how this simple drop-in replacement significantly improves the performance. With three button presses, the While Loop executes only 13 times (this number will vary, depending on how long you hold down the `Do some work?` button).

Figure 6.81. Wait For Front Panel Activity used in a While Loop to improve efficiency

[View full size image]



PREV NEXT

Wrap It Up!

LabVIEW has two structures to repeat execution of a subdiagramthe *While Loop* and the *For Loop.* Both structures are resizable boxes. 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 FALSE (or TRUE, depending on its configuration). The For Loop executes a specified number of times.

Shift registers, available for While Loops and For Loops, transfer values from the end of one loop iteration to the beginning of the next. You can configure shift registers to access values from many previous iterations. For each iteration you want to recall, you must add a new element to the left terminal of the shift register. You can also have multiple shift registers on a loop to store multiple variables.

LabVIEW has two structures to add control to data flowthe *Case Structure* and the *Sequence Structure.* Use of the Sequence Structure should be limited. Try to only use the Flat Sequence (not the Stacked Sequence) Structure, when possible.

You use the Case Structure to branch to different subdiagrams depending on the input to its selector terminal, much like an IF-THEN-ELSE structure in conventional languages (but more closely related to the CASE, SELECT, or SWITCH structures). Simply place the subdiagrams you want to execute inside the border of each case of the Case Structure and wire an input to the case selector terminal. Case Structures can be either Boolean (with two cases), numeric, or string (with up to 2^{15} 1 cases)LabVIEW automatically determines which type when you wire a Boolean, numeric, or string control to the selector terminal.

Sometimes the principles of dataflow do not cause your program to behave the way you want it to, and you need a way to force a certain execution order. The Flat Sequence Structure lets you set a specific order for your diagram functions. Use *tunnels* to pass values between Flat Sequence Structure frames.

Avoid using Stacked Sequence Structure so that you don't need to use *Sequence Locals* to pass values between frames (remember, Sequence Locals are evil). The data passed in a Sequence Local is available only in frames subsequent to the frame in which you created the Sequence Local or tunnel, NOT in those frames that precede the frame in which its value is assigned.

With the *Formula Node*, you can directly enter formulas in the block diagram, an extremely useful feature for concisely defining complex function equations. Remember that variable names are case sensitive and that each formula statement must end with a semicolon (;).

With the *Expression Node*, you can enter a single formula (expression) having one variable. The Expression Node uses the same syntax as the Formula Node.

The Programming>>Dialog & User Interface subpalette of the Functions palette provides functions that pop up dialog boxes. The One Button Dialog, Two Button Dialog, and Three Button Dialog functions pop up a dialog box containing the message of your choice. The Express Dialogs provide a quick way to present the user with information and prompt the user to enter simple data.

The Programming>>Timing subpalette of the Functions palette provides functions that control or monitor VI timing. The Wait (ms) function pauses your VI for the specified number of milliseconds. Wait Until Next ms Multiple can force loops to execute at a given interval by pausing until the internal clock equals (or has exceeded) a multiple of the millisecond input. These two wait functions are similar but not identical, and you don't really need to worry about the difference right now. Tick Count (ms) returns to you the value of the internal clock. In addition, LabVIEW provides with you two Express VIs for timing: Time Delay, which works just like Wait (ms), and Elapsed Time.

Timed Structures and VIs found on the Programming>>Structures>>Timed Structures palette allow you to control the rate and priority at which a timed structure executes its subdiagram, synchronize the start time of timed structures, create timing sources, and establish a hierarchy of timing sources. They only work on Windows and are designed specifically for LabVIEW RT and FPGA applications.

The While Loop + Case Structure combination is the work-horse of nearly every non-trivial LabVIEW application. This pattern will allow you to build simple yet powerful and scalable application frameworks.

PREV NEXT

Additional Activities

Activity 6-6: Equations

Build a VI that uses the Formula Node to calculate the following equations:

y1 = x3 + x2 + 5

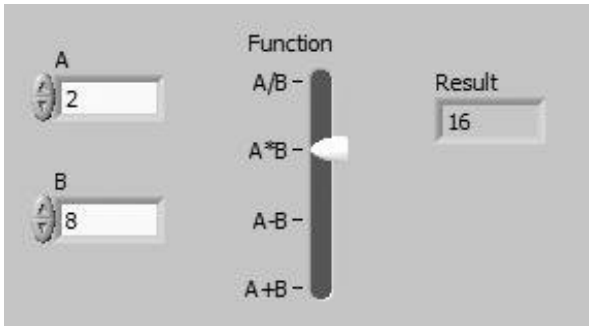
y2 = (m * x) + b

Use only one Formula Node for both equations. (Remember to put a semicolon [;] after each equation in the node.) Name the VI Equations.vi.

Activity 6-7: Calculator

Build a VI that functions like a calculator. The front panel should have digital controls to input two numbers, and a digital indicator to display the result of the operation (add, subtract, multiply, or divide) that the VI performs on the two numbers. Use a slide control to specify the operation to be performed (see Figure 6.82). Name the VI Calculator.vi.

Figure 6.82. Calculator.vi front panel, which you will build during this activity



You will want to use Text Labels on the slide control, obtained from the pop-up menu of the slide, to specify the function (add, subtract, multiply, and divide). If you don't see this option in the pop-up menu, make sure you're popping up on the slide itself and not the scale. Slides with text labels behave very much like text ring controls. When you first select Text Labels, the slide will have two settings, `max` *and* `min`*. You can use the Labeling tool to change this text. To add another text marker to the slide, pop up on the text display that appears next to your slide and select Add Item After or Add Item Before and then type in that marker's text. Note that wiring the output of a slide with text labels to the selector terminal of a Case Structure will not cause the cases to be typed as stringsthey will still be typed as numeric integers.*

Activity 6-8: Combination For/While Loop Challenge

Using only a While Loop, build a combination For Loop/While Loop that stops either when it reaches "N" (specified inside a front panel control), or when a user pushes a stop button. Name the VI Combo For-While Loop.vi.

Don't forget that a While Loop only executes while the conditional terminal (configured for Stop if True from its pop-up menu) reads a FALSE value. You might want to use the Or function (Programming>>Boolean subpalette of the Functions palette). Also, remember that while executing a loop, LabVIEW does not update indicators or read controls that are outside of the loop. Your `stop` *button MUST be inside your loop if you want correct functionality.*

Activity 6-9: Dialog Display

Write a VI that reads the value of a front panel switch, and then pops up a dialog box indicating if the switch is on or off. Name the VI Dialog Display.vi. If you've developed the bad habit of using the continuous run button, now is the time to break it, or you will get yourself stuck in an endless loop! If you do get stuck, use the keyboard shortcut to stop your VI: <control-.> under Windows,

<command-.> on a Macintosh, and <meta-.> under Linux.

PREV NEXT

7. LabVIEW's Composite Data: Arrays and Clusters

Overview

In this chapter, you will learn about two new, more complex data typesarrays and clusters. These composite data types allow you great flexibility in data storage and manipulation. You will also see some valuable uses for arrays and clusters and learn how you can use built-in functions to manage them. Finally, you will learn about how to work with errors in LabVIEW and good programming practices for working with error clusters.

Goals

- Learn about the built-in array manipulation functions
- Grasp the concept of polymorphism
- Learn how to use clusters, and how to bundle and unbundle them
- Understand how clusters differ from arrays
- Learn about error clusters and error handling

Key Terms

- Array
- Auto-indexing
- Polymorphism
- Cluster
- Bundle
- Unbundle
- Error cluster
- Error handling

What Are Arrays?

Until now, we've dealt with scalar numbers only (a *scalar* is simply a data type that contains a single value, or "non-array"), and now it's time to move onto something more powerful and compound. A LabVIEW *array* is a collection of data elements that are all the same type, just like in traditional programming languages. An array can have one or more dimensions, and up to 2^{31}-1 elements per dimension (memory permitting, of course). An array data element can have any type except another array, a chart, or a graph.

Array elements are accessed by their indices; each element's *index* is in the range 0 to N-1, where *N* is the total number of elements in the array. The *one-dimensional* (1D) array shown here illustrates this structure. Notice that the *first* element has index 0, the *second* element has index 1, and so on.

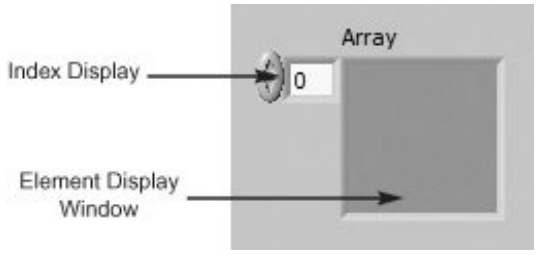
Index	0	1	2	3	4	5	6	7	8	9
10-Element Array	12	32	82	8.0	4.8	5.1	6.0	1.0	2.5	1.7

You will find that waveforms (and many other things) are often stored in arrays, with each point in the waveform comprising an element of the array. Arrays are also useful for storing data generated in loops, where each loop iteration generates one element of the array.

Creating Array Controls and Indicators

It takes two steps to make the controls and indicators for compound data types like arrays and clusters. You create the array control or indicator by combining an *array shell* with a *data object*, which can be numeric, Boolean, path, or string (or cluster, but we'll cover that later). You will find the array shell in the Modern>>Array, Matrix, & Cluster subpalette of the Controls palette (see Figure 7.1).

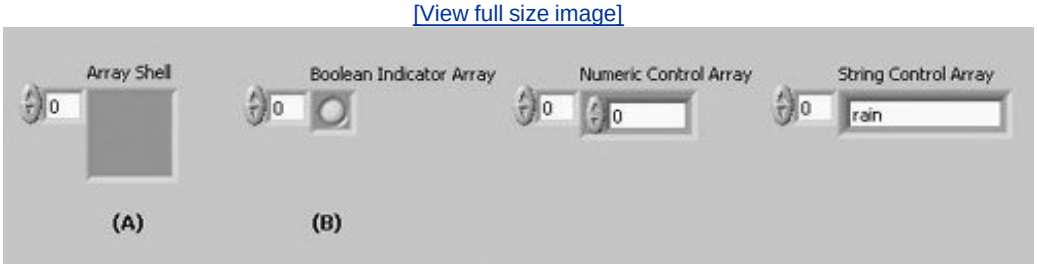
Figure 7.1. Array shell



To create an array, drag a data object into the element display window. You can also deposit the object directly by clicking inside the window when you first choose the object from the Controls palette. The element display window resizes to accommodate its new data type, as shown in Figure 7.2, but remains grayed out until you enter data into it. Note that all elements of an array must be either controls or indicators, not a combination.

Figure 7.2. Front panel showing an empty array shell (A) and three populated array shells (B)

[View full size image]

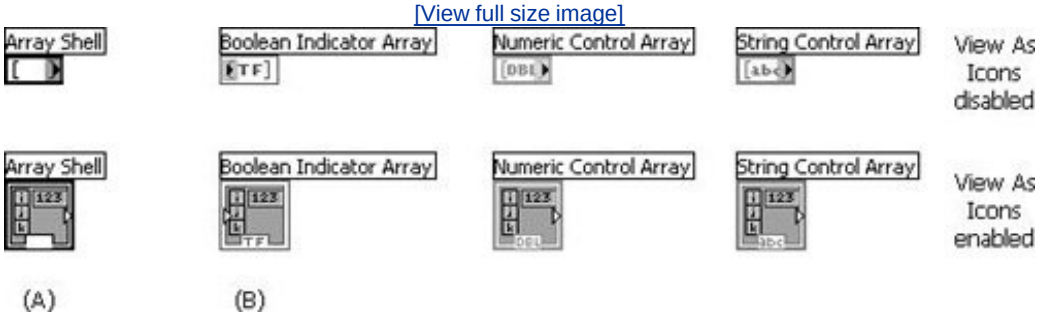


When you first drop an array shell on the front panel, its block diagram terminal is black, characteristic of an undefined data type. The terminal also contains brackets (when View As Icon

terminal option disabled) or an array icon (when View As Icon is enabled), shown in part (A) of Figure 7.3, which are LabVIEW's way of denoting an array structure. When you assign a data type to the array (by placing a control or indicator in its element display window), then the array's block diagram terminal assumes its new type's color and lettering, as in part (B). You will notice that array wires are thicker than wires carrying a single value.

Figure 7.3. Block diagram showing empty (A) and populated (B) array shell terminals

[View full size image]



You can enter data into your array as soon as you assign a data type to it. Use the Labeling or Operating tool to type in a value, or if your data is numeric, click the arrows of the index display to increment or decrement it.

If you want to resize the array element object, as shown in Figure 7.4, use the Positioning tool and make sure it turns into the resizing handles when you place it on the edge of the object (you will probably have to position it slightly inside the array shell, and you may have to move it around a bit to get the object inside the array shell to show the resizing handles). If you want to show more elements at the same time, as shown in Figure 7.5, move the Positioning tool around the edge of the array shell until the array shell shows the resizing handles, then stretch either horizontally or vertically (your data is unchanged by the layout of your array). You will then have multiple elements visible. The element closest to the index display always corresponds to the element number displayed there.

Figure 7.4. Resize array element

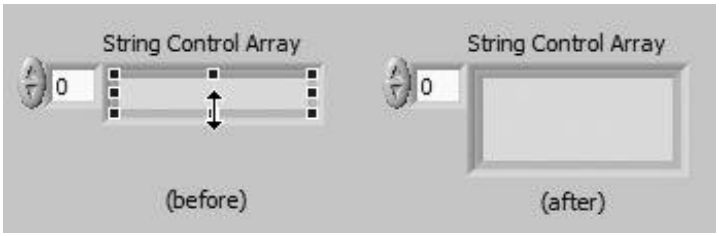
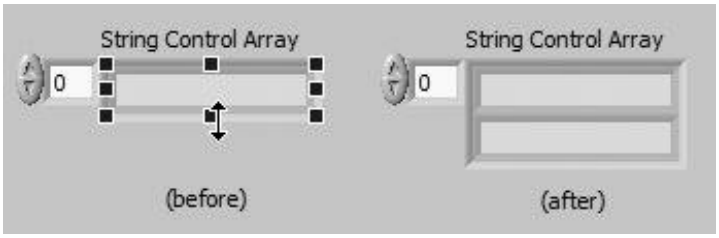


Figure 7.5. Resize array shell



You can create array constants on the block diagram just like you can create numeric, Boolean, or string constants. Choosing Array Constant from the Array subpalette of the Functions palette will give you an array shell; then simply place in an appropriate data type (usually another constant) just like you do on the front panel. This feature is useful when you need to initialize shift registers or provide a data type to a file or network function (which you'll learn about later).

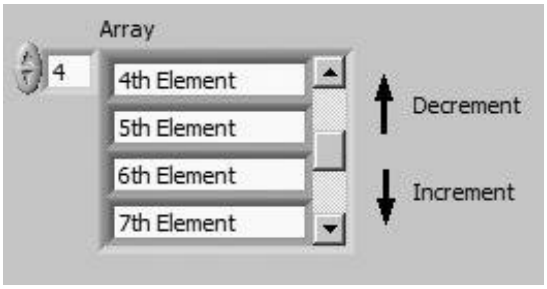
If you want to clear an array control, indicator, or constant of data, pop up on the *index display* (NOT the element itself or you'll get the wrong menu) and choose Data Operations>>Empty Array.

If you want to insert or delete an element from an array control, indicator, or constant, pop up on an *element of the array* and choose Data Operations>>Insert Element Before or Data Operations>>Delete Element.

Array Scrollbars

While changing the value in the index display allows you to change the array elements that are visible in the array shell, it is probably easier and more intuitive to use array scrollbars (see Figure 7.6). You can make the array shell scrollbars visible by right-clicking on the array shell and selecting Visible Items>>Vertical Scrollbar or Visible Items>>Horizontal Scrollbar from the shortcut menu.

Figure 7.6. Array scrollbars



PREV NEXT

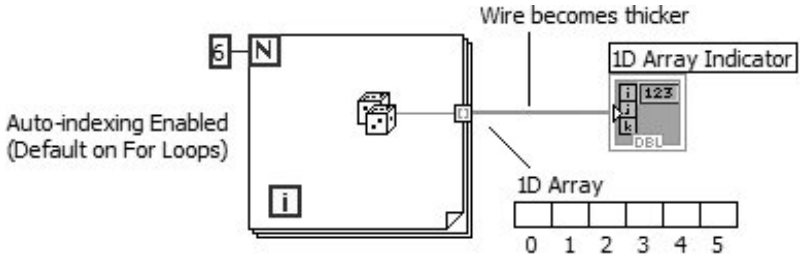
Using Auto-Indexing

The For Loop and the While Loop can index and accumulate arrays at their boundaries automaticallyone new element for each loop iteration. This capability is called *auto-indexing.*

One important thing to remember is that auto-indexing is enabled by default on For Loops but disabled by default on While Loops.

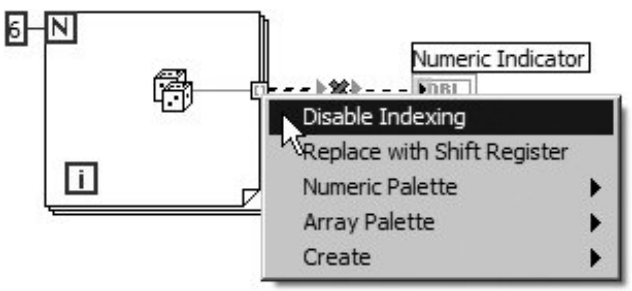
Figure 7.7 shows a For Loop auto-indexing an array at its boundary. Each iteration creates the next array element. After the loop completes, the array passes out of the loop to the indicator; none of the array data is available until after the loop finishes. Notice that the wire becomes thicker as it changes to an array wire type at the loop border.

Figure 7.7. A For Loop auto-indexing an array at is boundary



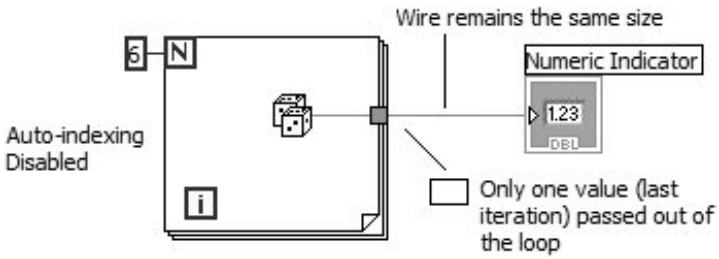
If you need to wire a scalar value out of a For Loop without creating an array, you must disable auto-indexing by popping up on the tunnel (the square with [ ] symbol) and choosing Disable Indexing from the tunnel's pop-up menu (see Figure 7.8).

Figure 7.8. Choosing Disable Indexing on a tunnel's pop-up menu



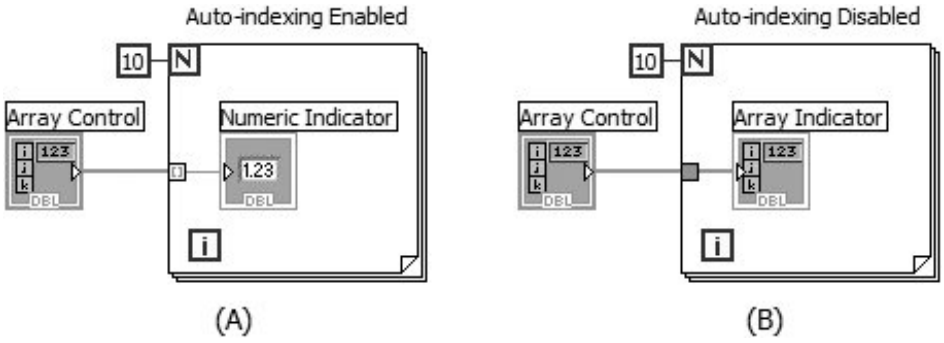
In Figure 7.9, auto-indexing is disabled, and only the last value returned from the Random Number (0-1) function passes out of the loop. Notice that the wire remains the same size after it leaves the loop. Pay attention to this wire size, because auto-indexing is a common source of problems among beginners. They often create arrays when they don't want to, or don't create them when they need them, and then go crazy trying to figure out why they get a bad wire.

Figure 7.9. A loop output tunnel with auto-indexing disabled



Auto-indexing also applies when you are wiring arrays into loops. If indexing is enabled as in loop (A), the loop will index off one element from the array each time it iterates (note how the wire becomes thinner as it enters the loop). If indexing is disabled as in loop (B), the entire array passes into the loop at once (see Figure 7.10).

Figure 7.10. An input tunnel with auto-indexing enabled (A) and disabled (B)

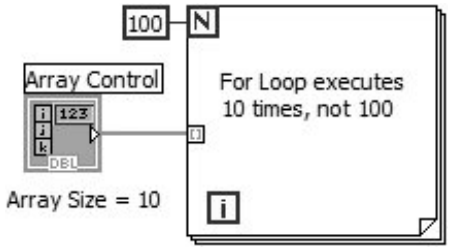


Because For Loops are often used to process arrays, LabVIEW enables auto-indexing by default when you wire an array into or out of them. By default, LabVIEW does NOT enable auto-indexing for While Loops. You must pop up on the array tunnel and choose Enable Indexing from the pop-up menu if you want your While Loop to auto-index. Pay close attention to the state of your indexing, lest you develop errors that are tricky to spot.

Using Auto-Indexing to Set the For Loop Count

When you enable auto-indexing on an array *entering* a For Loop, LabVIEW automatically sets the *count* to the array size, thus eliminating the need to wire a value to the count terminal. If you give LabVIEW conflicting countsfor example, by setting the count explicitly and by auto-indexing (or by auto-indexing two different size arrays)LabVIEW sets the count to the smallest of the choices. In Figure 7.11, the array size, and not the value wired to the count terminal, determines the number of For Loop iterations, because the array size is the smaller of the two.

Figure 7.11. A For Loop that will execute 10 times

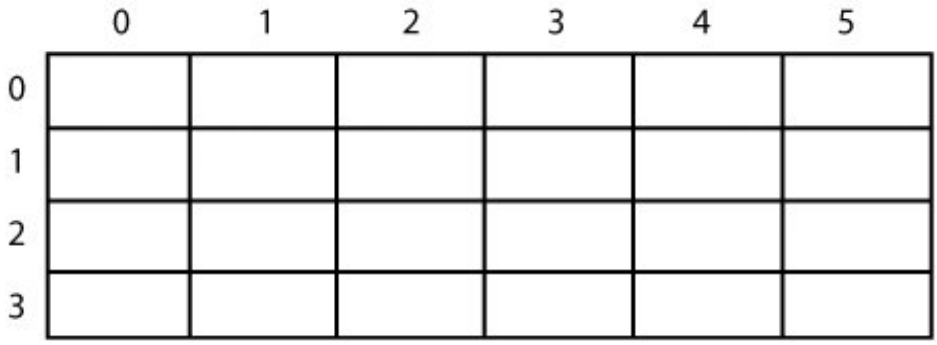


PREV NEXT

Two-Dimensional Arrays

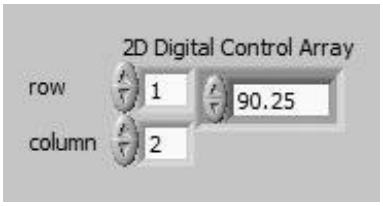
A two-dimensional, or 2D, array stores elements in a grid-like fashion. It requires two indices to locate an element: a column index and a row index, both of which are zero-based like everything else in LabVIEW. Figure 7.12 shows how a six-column by four-row array that contains six times four elements is represented.

Figure 7.12. Six-column by four-row array of 24 elements



You can add dimensions to an array control or indicator by popping up on its *index display* (not on the element display) and choosing Add Dimension from the pop-up menu. Figure 7.13 shows a 2D array of digital controls. Notice that you now have two indices to specify each element. You can use the Positioning tool to expand your element display in two dimensions so that you can see more elements, by hovering the mouse over the array shell until the resizing handles appear and then dragging the resizing handles.

Figure 7.13. 2D array of digital controls



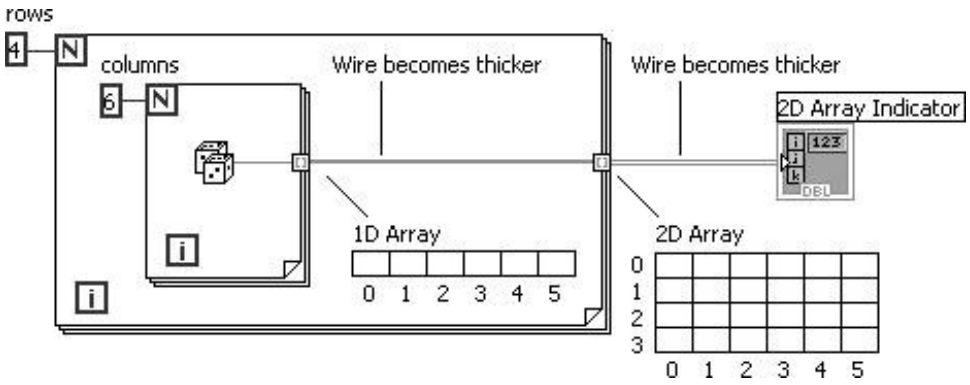
Remove unwanted dimensions by selecting Remove Dimension from the index display's pop-up menu.

Creating Two-Dimensional Arrays

You can use two For Loops, one inside the other, to create a 2D array if you don't want to type in values on the front panel. The inner For Loop creates a row, and the outer For Loop "stacks" these rows to fill in the columns of the matrix. The illustration in Figure 7.14 shows two For Loops creating a 2D array of random numbers using auto-indexing.

Notice that a 2D array wire is even thicker than a 1D array wire.

Figure 7.14. Two For Loops creating a 2D array using auto-indexing



PREV NEXT

Activity 7-1: Building Arrays with Auto-Indexing

Now we'll give you a chance to better understand arrays and auto-indexing by working with them yourself. In this activity, you will open and observe a VI that uses auto-indexing on both a For Loop and a While Loop to create arrays of data.

1. Open the Building Arrays.vi example, located in `EVERYONE\CH07`. This exercise generates two arrays on the front panel, using a For Loop to create a 2D array and a While Loop to create a 1D array. The For Loop executes a set number of times; you must hit the `STOP` button to halt the While Loop (or it will stop after 101 iterations).

2. Take a look at the front panel (see Figure 7.15), and then switch to the block diagram (see Figure 7.16). Notice how the nested For Loops create a 2D array's rows and columns at their borders, respectively, using auto-indexing. Also notice how the auto-indexed wires become thicker as they leave the loop boundaries.

Figure 7.15. Building Arrays.vi front panel

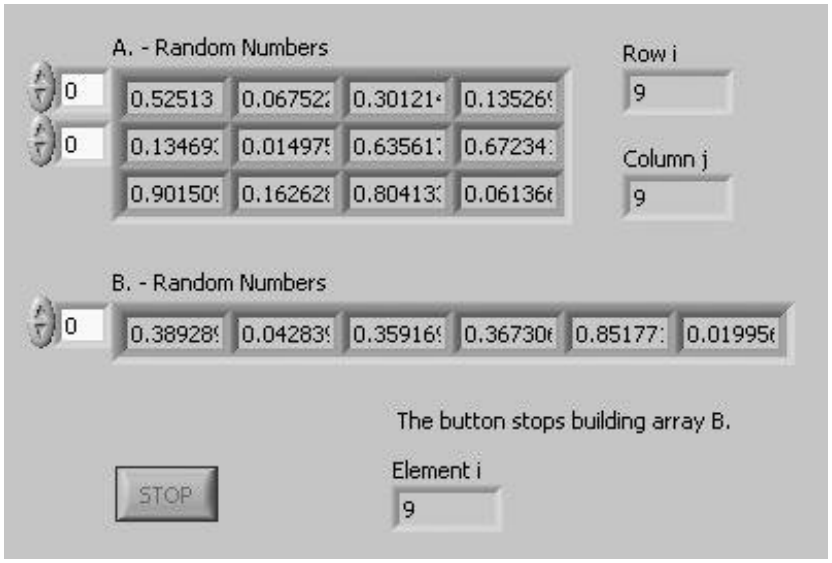
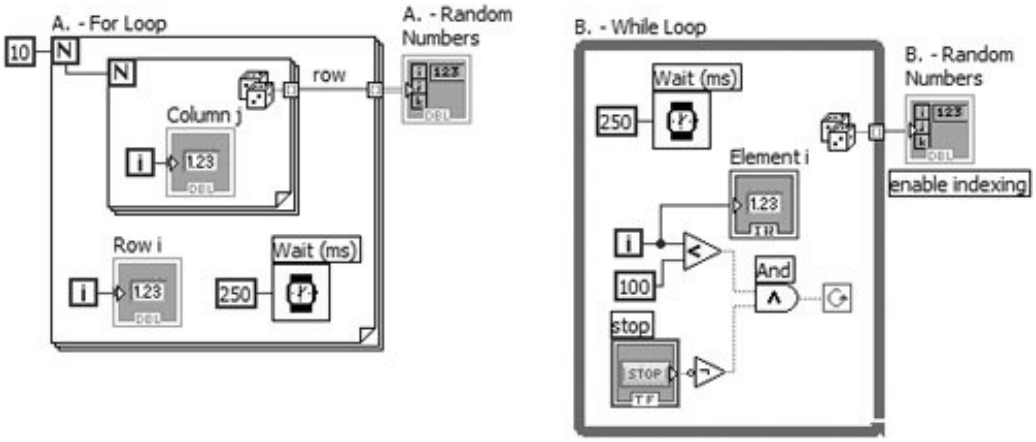


Figure 7.16. Building Arrays.vi block diagram

[View full size image]



3. Before we could get array data out of the While Loop, we had to pop up on the tunnel containing the random number and Select Enable Indexing. To see how this works, go ahead and pop up on the tunnel, then select Disable Indexing. You will see the wire leaving the Loop break. Pop up on the tunnel again and select Enable Indexing to fix it.

   This loop uses a little logic algorithm to ensure that if the user does not press the STOP button after a reasonable amount of time (101 iterations), the loop stops anyway. If the user has not pressed the STOP button AND the loop has executed fewer than 101 times, the loop continues. If either of those conditions changes, the loop stops.

   Why does it execute 101 times, instead of 100? Remember that the While Loop checks the conditional terminal at the end of each iteration. At the end of the 100th iteration, i 99 (because it starts at zero), which is less than 100, and the loop continues. At the end of the 101st iteration, the count is no longer less than 100 and the loop stops (assuming the STOP button hasn't already stopped it).

4. Now that you understand how it works, run the VI. Remember to hit STOP to halt the While Loop, especially because the front panel indicator does not update until the entire array has been generated (remember, controls and indicators outside loops are not read or updated while the loop is executing).

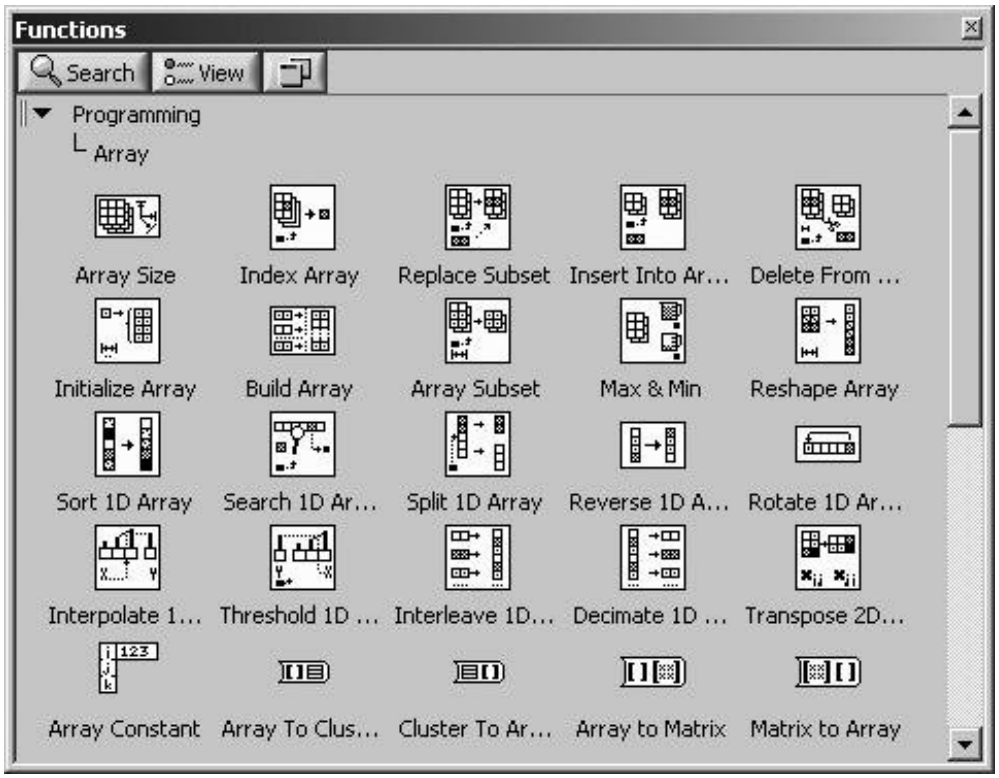
5. Close the VI and don't save any changes.

PREV NEXT

Functions for Manipulating Arrays

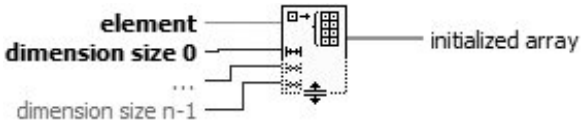
By now, you know a little bit about what arrays are, so let's talk about all the cool stuff you can do with them. LabVIEW has many functions to manipulate arrays in the Programming>>Array subpalette of the Functions palette. To avoid a common pitfall, always keep in mind that arrays (and all other LabVIEW structures) are zero indexedthe first element has an index of zero, the second has an index of one, and so on. Some common functions are discussed here, but you might also want to browse through the Array subpalette, shown in Figure 7.17, just to see what else is built in for you.

Figure 7.17. Array palette



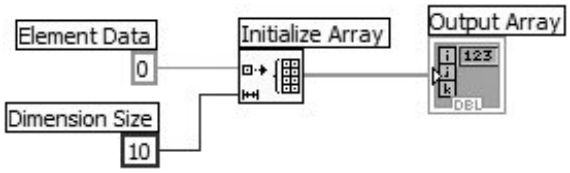
- Initialize Array will create and initialize an n-dimensional array with the value of your choice (see Figure 7.18). You can configure it for larger dimensions by "growing" it with the Positioning tool to provide more dimension size inputs. This function is useful for allocating memory for arrays of a certain size or for initializing shift registers with array-type data.

Figure 7.18. Initialize Array



In Figure 7.19, Initialize Array shows how to initialize a ten-element, one-dimensional array of DBL numerics, with each element of the array containing a zero.

Figure 7.19. Initialize Array function used to initialize an array



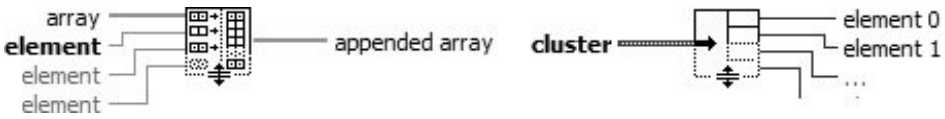
- Array Size returns the number of elements in the input array. If the input array is n-dimensional, Array Size returns an *n*-element, one-dimensional array, with each element containing the size of one of the array's dimensions.
- Depending on how you configure it, Build Array concatenates, or combines, two arrays into one, or adds extra elements to an array. The function looks like the icon at left when first placed in the diagram window. You can resize or "grow" this function to increase the number of inputs. Build Array has two types of inputs, *array* and *element*, so that it can assemble an array from both array and single-valued inputs (see Figure 7.22).

Build Array

Figure 7.20. Array Size

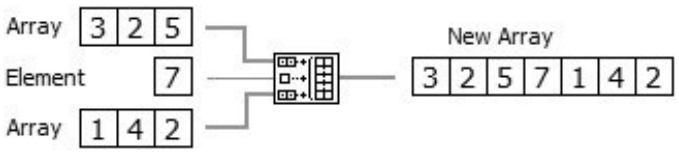
Figure 7.21. Array size used to determine the length of a four-element 1D array

Figure 7.22. Build Array



For example, the Build Array function shown in Figure 7.23 has been configured to concatenate two arrays and one element into a new array.

Figure 7.23. Build Array used to concatenate a 1D array, a scalar, and another 1D array



The Build Array function input will automatically adapt to an element or an array input, depending on what you wire to it.

Pay special attention to the inputs of the Build Array function. Array inputs have two "dotted" squares, while element inputs have a single hollow square. Although LabVIEW will adapt the input type depending on what kind of data you wire to it (elements or arrays), they are NOT interchangeable and can cause lots of confusing bad wires if you're not careful.

As you get more advanced, you'll find that Build Array can also build or add elements to multidimensional arrays. To add an element to a multidimensional array, the element must be an array of one size smaller dimension (i.e., you can add a 1D element to a 2D array). You can also build a 2D array by using the Build Array function and wiring 1D arrays in as "elements" (each 1D array will become one row of the 2D array). Sometimes you'll want to concatenate several 1D arrays together, instead of building a 2D array. In this case, you'll need to pop up on the Build Array function, and choose Concatenate Inputs.

- Array Subset returns a portion of an array starting at index and containing length elements (see Figure 7.24). Notice that the third element's index is two because the index starts at zero; that is, the *first element has an index of zero* (see Figure 7.25).

Figure 7.24. Array Subset

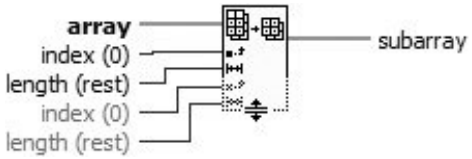
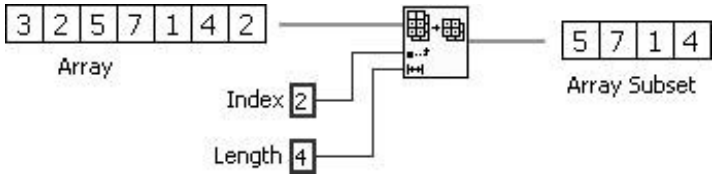


Figure 7.25. Array Subset used to obtain a subset of four elements starting at index two



- Index Array (see Figure 7.26) accesses a particular element of an array. An example of the Index Array function accessing the third element of an array is shown in Figure 7.27.

Figure 7.26. Index Array

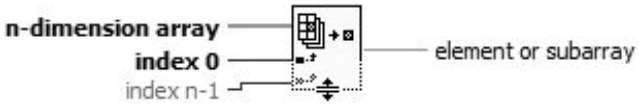
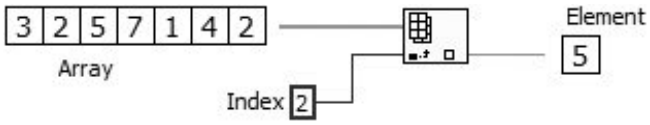


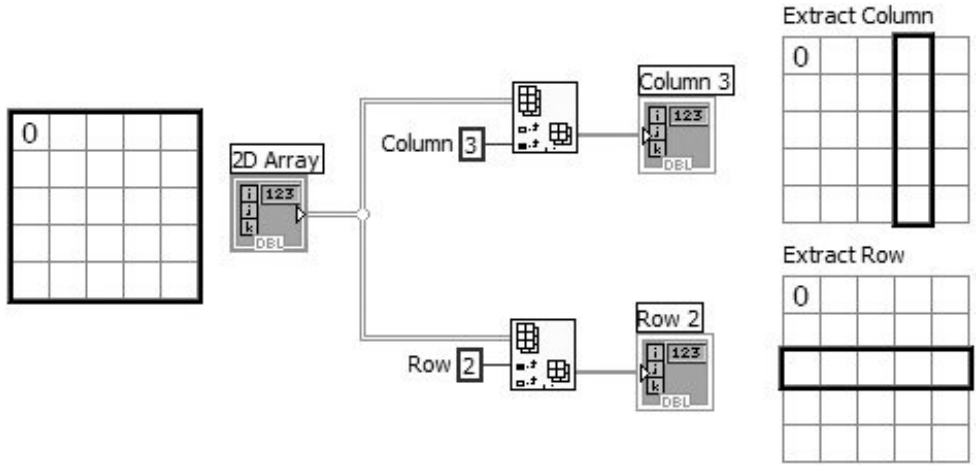
Figure 7.27. Index Array used to obtain the third element of a 1D array



You also can use this function to *slice off* a row, column, or scalar element of a 2D array. To extract a single scalar element, wire the desired element's row index to the top input and its column index to the bottom input. To extract a row or column, you simply leave one of the Index Array function inputs unwired. To slice off a row from the 2D array, wire the row index (the first index input) of the row you want to slice off. To slice off a column from the 2D array, leave the first index input unwired, and wire the column index (the second index input) with the column you want to slice off.

Notice that the index terminal symbol changes from a solid to an empty box when you leave its input unwired. The illustration in Figure 7.28 shows how to extract a column or a row from a 2D array.

Figure 7.28. Extracting a column and row from a 2D array



- Delete From Array deletes a portion of an array starting at index and containing length elements (see Figure 7.29). Similar to the Array Subset function, Delete From Array returns subset of an array, but it also returns the original array minus the deleted subset (see Figure 7.30).

Figure 7.29. Delete From Array

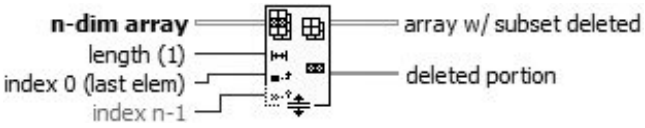
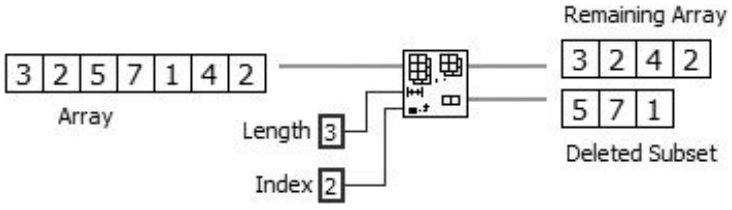


Figure 7.30. Delete From Array used to delete a subset of three elements starting at index two



Notice that the length and index terminals are interchanged between the Array Subset and Delete From Array functions.

PREV NEXT

Activity 7-2: Array Acrobatics

Confused yet? You'll get a better feel for arrays as you work with them. In this exercise, you will finish building a VI that concatenates two arrays and then indexes out the element in the middle of the new concatenated array.

1. Open Array Exercise.vi, located in `EVERYONE\CH07`.

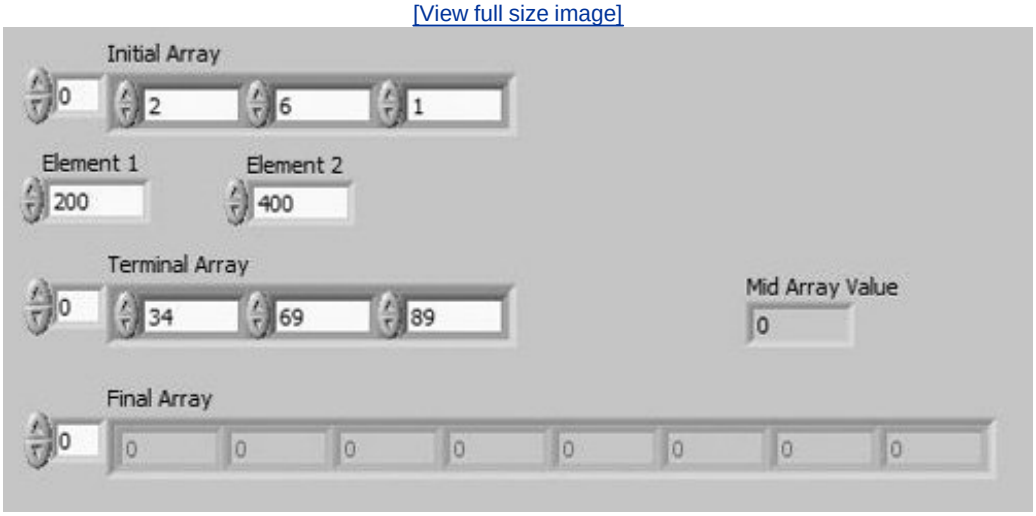
   The front panel contains two input arrays (each showing three elements), two digital controls, and an output array (showing eight elements). The VI will concatenate the arrays and the control values in the following sequence to create the new array.

   Initial Array + Element 1 + Element 2 + Terminal Array

   The front panel is already built. You will finish building the diagram. Note that an array control or indicator will appear grayed out until you or the program assigns it some data (see Figure 7.31).

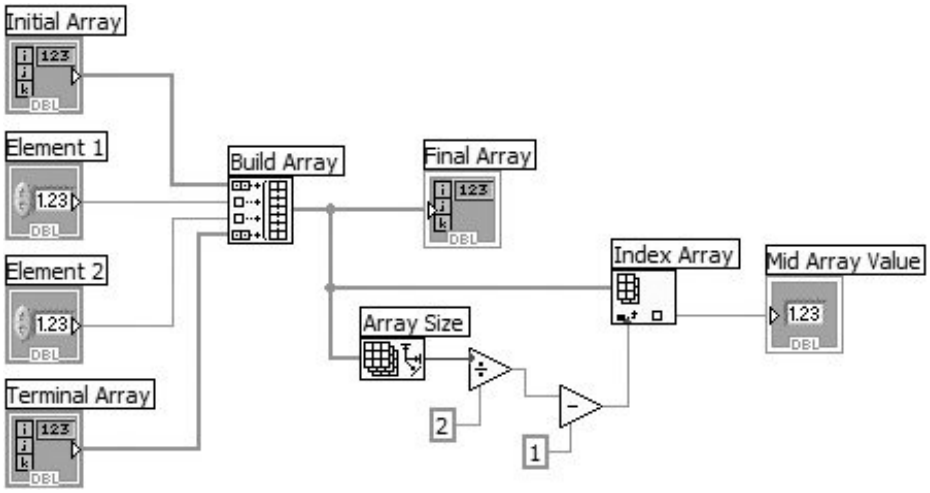
Figure 7.31. Array Excersise.vi front panel

[View full size image]



2. Build the block diagram in Figure 7.32. Use the Help window to find the correct terminals on the functions.

Figure 7.32. Array Exercise.vi block diagram



Build Array Function

Build Array function (Programming>>Array palette). In this exercise, concatenates the input data to create a new array in the following sequence: Initial Array + Element 1 + Element 2 + Terminal Array.

Build Array Function

The function looks like the icon at the left when placed in the diagram window. Place the Positioning tool on the lower-middle edge and resize the function with the resize handles until it includes four inputs.

Array Input

Element Input

The inputs will automatically detect whether you are wiring an array or an element. The symbols at the left indicate whether an input is an array or an element.

Array Size Function

Array Size function (Programming>>Array palette). Returns the number of elements in the concatenated array.

Index Array Function

Index Array function (Programming>>Array palette). In this exercise, returns the element in the middle of the array.

LabVIEW builds the array using the Build Array function, and then calculates the index for the middle element of the array by taking the length of the array, dividing it by two, and subtracting one (to account for the zero-based array index). Because the array has an even number of elements, the middle element will actually be one of the two middle elements.

Notice those coercion dots? You can't get rid of them simply by changing the two constants to I32s, because the Divide function coerces them back in order to handle fractional outputs. But if you replace the Divide

with the Quotient & Remainder (right next to it on the Numeric palette), the dots will go away. Because the Quotient output is a floor function, you no longer need the Subtract.

3. Return to the front panel and run the VI. Try several different number combinations.

4. Save the finished VI in your MYWORK directory and close the VI.

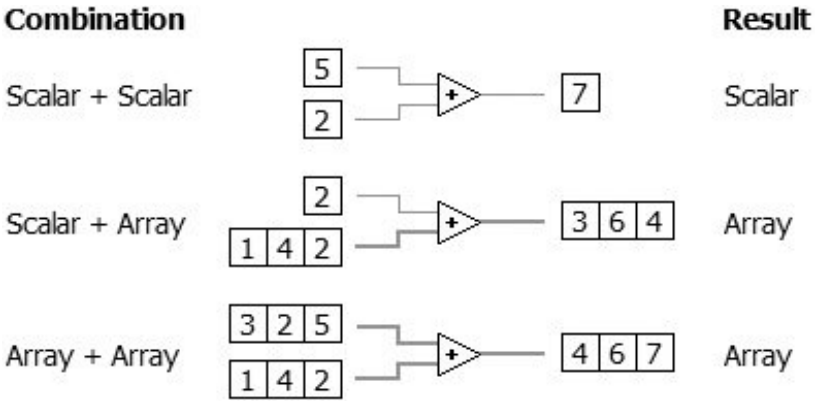
You might also want to look through the programs in EXAMPLES/GENERAL/ARRAYS.LLB in the LabVIEW directory, to see some other things you can do with arrays. These examples can be easily found using the NI Example Finder (from the Help>>Find Examples menu) by selecting Browse according to: Directory Structure and then navigating the tree control to general/arrays.llb.

PREV NEXT

Polymorphism

Another handy feature you'll find useful is the *polymorphism* of the LabVIEW arithmetic functions, Add, Multiply, Divide, and so on. *Polymorphism* is just a big word for a simple principle: the inputs to these functions can be of different size and representation. For example, you can add a scalar to an array or add two arrays together using the same function. Figure 7.33 shows some of the polymorphic combinations of the Add function.

Figure 7.33. Some of the polymorphic combinations of the Add function



In the first combination, the result is a scalar number. In the second combination, the scalar is added to *each* element of the array. In the third combination, each element of one array is added to the corresponding element of the other array. The same Add function is used in all instances, but it performs a different type of operation in each.

In Figure 7.34, each iteration of the For Loop generates one random number (valued between 0 and 1) that is stored in the array created at the border of the loop. After the loop finishes execution, the Multiply function multiplies each element in the array by the scaling factor you set. The front panel array indicator then displays the scaled array.

Figure 7.34. Multiplying a 1D numeric array by a scalar numeric value

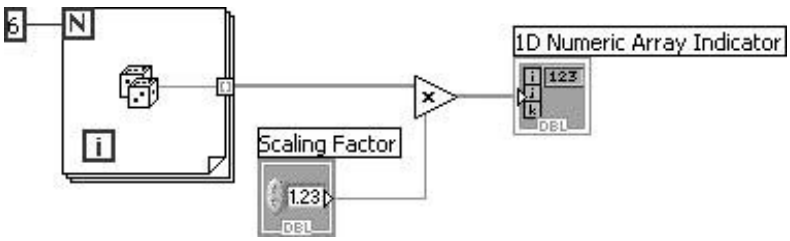
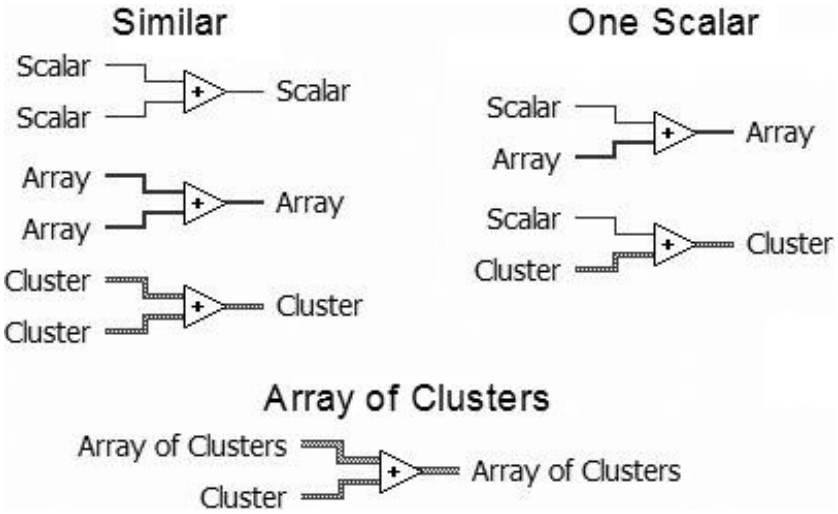


Figure 7.35 shows some of the possible polymorphic combinations of the Add function. You'll learn about the clusters it depicts in the next section.

Figure 7.35. Some possible polymorphic combinations of the Add function



If you are doing arithmetic on two arrays with a different number of elements, the resulting array will be the size of the smaller of the two. In other words LabVIEW operates on corresponding elements in the two arrays until one of the arrays runs out of elements. The remaining elements in the longer array are ignored.

In addition to functions that are polymorphic, VIs can be polymorphic too. A polymorphic VI is a special type of VI that is actually a group of VIs, each VI handling a different data type. You can even create your own polymorphic VIs. You will learn more about polymorphic VIs in Chapter 13, "Advanced LabVIEW Structures and Functions."

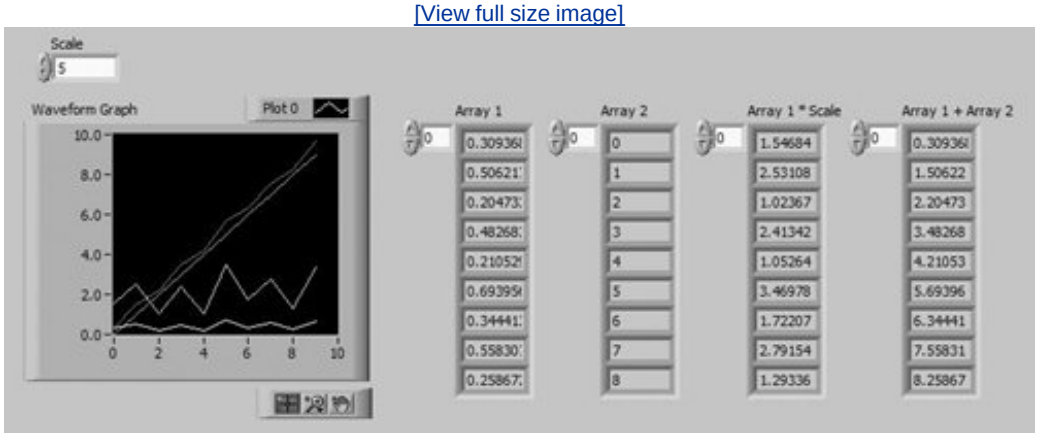
PREV NEXT

Activity 7-3: Polymorphism

You will build a VI that demonstrates polymorphism on arrays (see Figure 7.36).

Figure 7.36. Front panel of the VI you will create during this activity

[View full size image]



1. Open a new VI and create the front panel shown in Figure 7.36.

   First, create the arrays. Remember, to create an array, you must first select the Array shell from the Modern>>Array, Matrix & Cluster subpalette of the Controls palette. Then put a numeric indicator into the shell's data object window. To see more than one element in the array, you must grab and drag the resizing handles of the filled element display window with the Positioning tool. You can drag out multiple elements in a 1D array either horizontally or vertically.

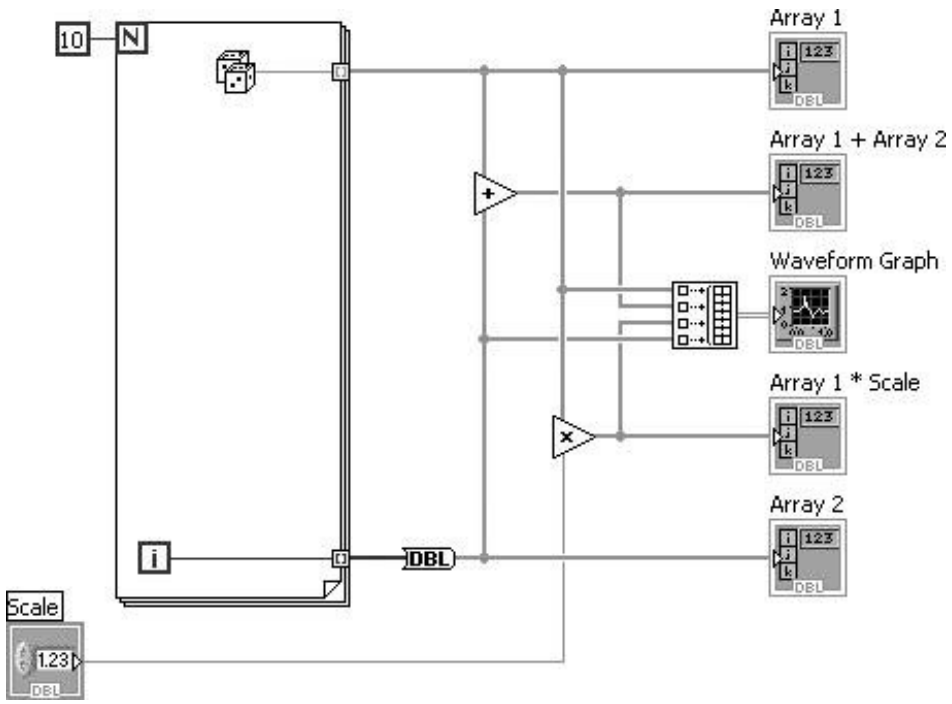
   All four arrays in this exercise contain indicator elements. Make sure to give them unique labels so that you don't get them confused. If you ever do lose track of which front panel object corresponds to which block diagram terminal, simply pop up on one of them and choose Find Terminal or Find Indicator; LabVIEW will highlight the corresponding object.

2. After you've created the arrays, select a Waveform Graph from the Graph subpalette of the Controls palette. Although you'll learn more about graphs in the next chapter, we wanted to give you a preview and spice up this exercise.

3. Don't forget to create the `Scale` control.

4. Build the diagram shown in Figure 7.37be careful because the wiring can get a little tricky!

Figure 7.37. Block diagram of the VI you will create during this activity



Auto-indexing is enabled by default on the For Loop, so your arrays will be generated automatically.

5. You can find Add, Multiply, and Random Number (0-1) in the Programming>>Numeric palette.

6. Select Build Array from the Programming>>Array palette. You will have to grow it using the Positioning tool so that it has four inputs. By default, the array inputs are not concatenated; the output from Build Array is a 2D array. Each input array becomes a row, so the output 2D array is composed of four rows and ten columns.

7. Run the VI. Your graph will plot each array element against its index for all four arrays at once: `Array 1` data, `Array 2` data, `Array 1* Scale`, and `Array 1+ Array 2.`

   The results demonstrate several applications of polymorphism in LabVIEW. For example, `Array 1` and `Array 2` could be incoming waveforms that you wish to scale.

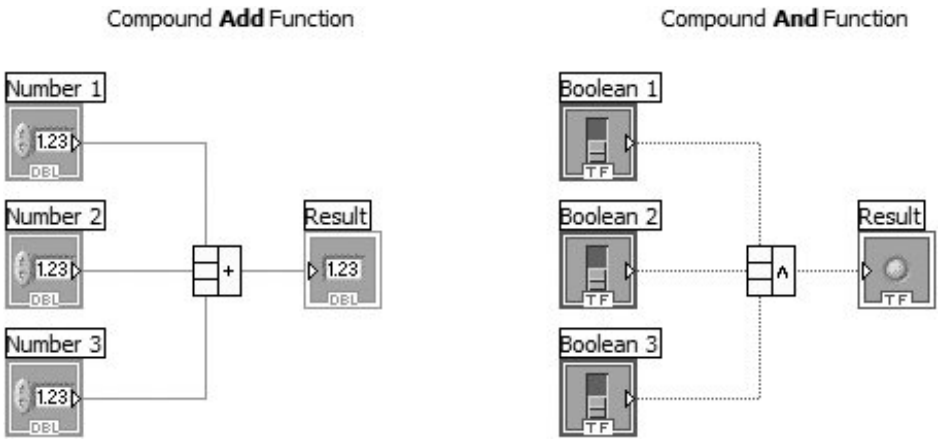
8. Save the VI as Polymorphism Example.vi and place it in your `MYWORK` directory or VI library. Close the VI.

PREV NEXT

Compound Arithmetic

While we're on the subject of arithmetic, we should mention the Compound Arithmetic function (see Figure 7.38). While the previous polymorphism example showed you how to add and multiply different-sized data together, this function lets you operate on more than two numbers simultaneously. The Compound Arithmetic function eliminates the need for multiple Add, Multiply, AND, OR, and XOR terminals should you need to perform one of these functions on several numbers at once (AND, OR, and XOR are Boolean arithmetic operations).

Figure 7.38. (Left) Compound Add Function (Right) Compound And Function

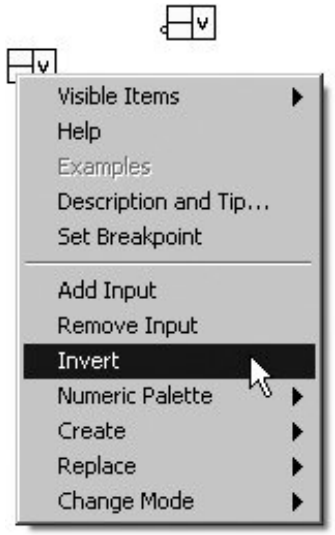


The Compound Arithmetic function can be found in both the Numeric and Boolean subpalettes of the Functions palette. Like many other functions, you can use the Positioning tool to grow it and provide more input terminals. The Compound Arithmetic function has only one form; you configure it to perform the arithmetic function of your choice. To change the function (choices are Add, Multiply, AND, OR, or XOR), pop up on the output terminal and select Change Modes. You can also click on the function with the Operating tool to change the mode.

Select the Invert pop-up option to invert the sign of numeric inputs and outputs or the value of Booleans (from FALSE to TRUE or vice versa), as shown in Figure 7.39. A small circle at the input or output symbolizes an inverted value.

Figure 7.39. Selecting the Invert option from the pop-up menu of one of

the terminals of the Compound Arithmetic function.



The Invert option has a different effect depending on the Compound Arithmetic function's mode of operation (as selected from the Change Mode pop-up submenu). Table 7.1 shows the effect of the invert option, applied to an *input*, for each of the Compound Arithmetic function's modes of operation.

Table 7.1. The Effect of an Inverted Input for Each Compound Arithmetic Mode

Mode	*Non-Inverted Input*	*Inverted Input*
Add	add input	add (0input)
Multiply	multiply input	multiply (1/input)
AND	AND input	AND (NOT input)
OR	OR input	OR (NOT input)
XOR	XOR input	XOR (NOT input)

When the Invert option is applied to an *output*, the function is similar. Table 7.2 shows the effect of the invert option, applied to an *output*, for each of the Compound Arithmetic function's modes of operation.

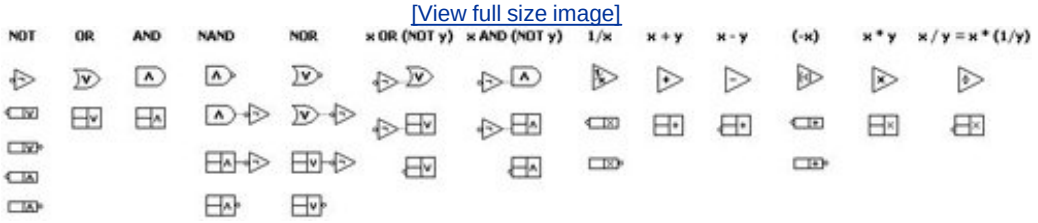
Table 7.2. The Effect of an Inverted Output for Each Compound Arithmetic Mode

Mode	*Non-Inverted Output*	*Inverted Output*
Add	result	0result
Multiply	result	1/result
AND	result	NOT result (NAND)
OR	result	NOT result (NOR)
XOR	result	NOT result (NOT XOR)

In order to make this as clear as possible, we have created a collection of equivalent operations, shown in Figure 7.40. This is meant only to help you see the underlying principles. (Don't worry; this won't be on the quiz.)

Figure 7.40. Equivalent standard and compound arithmetic operations (in columns)

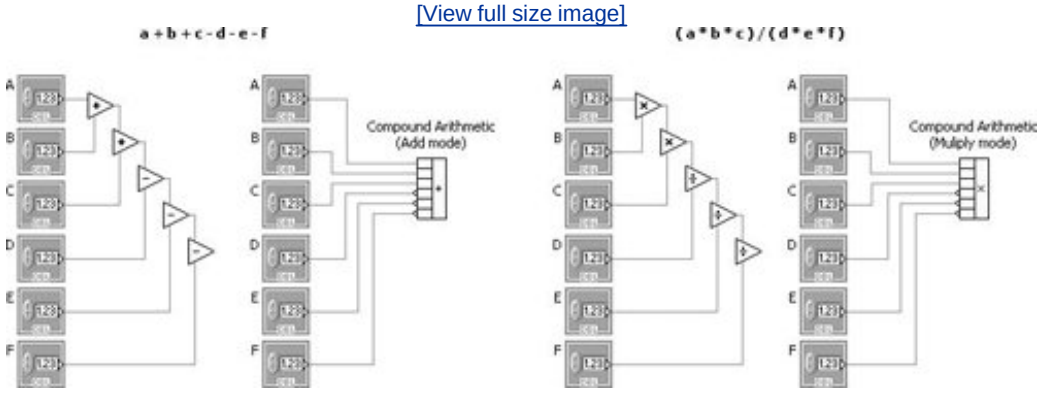
[View full size image]



For a simpler example of some practical ways to use the compound arithmetic node, see Figure 7.41. The two equivalent operations on the left show how you can use the compound arithmetic node for combined addition and subtraction. The two equivalent operations on the right show how you can use the compound arithmetic node for combined multiplication and division.

Figure 7.41. Equivalent standard and compound arithmetic operations (in columns)

[View full size image]



A Word About Boolean Arithmetic

LabVIEW's Boolean arithmetic functionsAnd, Or, Not, Exclusive Or, Not Exclusive Or, Not And, and Not Orcan be very powerful. You will see occasional Boolean arithmetic throughout this book. If you've never seen it before, we recommend reading about it in a good logic or digital design book. But just as a refresher, we'll mention a few basics. If you can't remember which function does what, use the Help window!

Not is probably the easiest to describe, because it simply inverts the input value. If the input value is TRUE, Not will output FALSE; if the input is FALSE, Not returns TRUE.

The And function outputs a TRUE only if all inputs are TRUE.

The Or function outputs a TRUE if at least one input is TRUE.

The And and Or functions have the following outputs, given the inputs shown:

FALSE And FALSE = FALSE

TRUE And FALSE = FALSE

FALSE And TRUE = FALSE

PREV NEXT

All About Clusters

Now that you've got the concept of arrays under your belt, clusters should be easy. Like an array, a *cluster* is a data structure that groups data. However, unlike an array, a cluster can group data of different types (i.e., numeric, Boolean, etc.); it is analogous to a *struct* in C or the data members of a *class* in C++ or Java. A cluster may be thought of as a *bundle* of wires, much like a telephone cable. Each wire in the cable represents a different element of the cluster. Because a cluster has only one "wire" in the block diagram (even though it carries multiple values of different data types), clusters reduce wire clutter and the number of connector terminals that subVIs need. You will find that the cluster data type appears frequently when you plot your data on graphs and charts. Figures 7.42 and 7.43 illustrate the concepts of bundling and unbundling clusters of element data.

Figure 7.42. A conceptual illustration showing the bundling of data elements to create a cluster

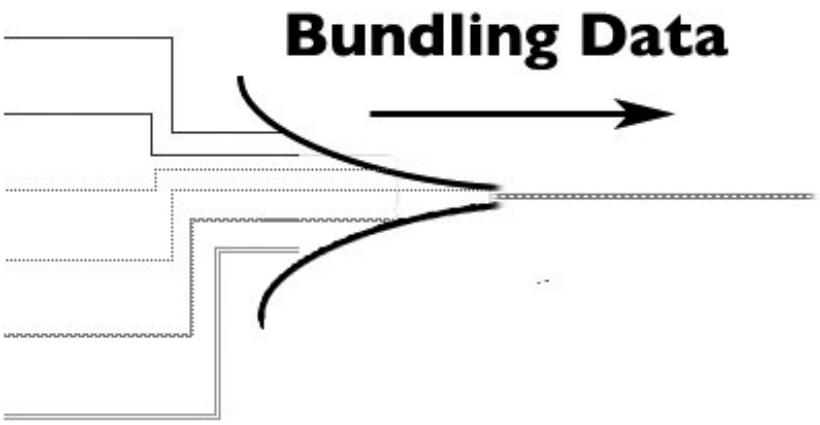
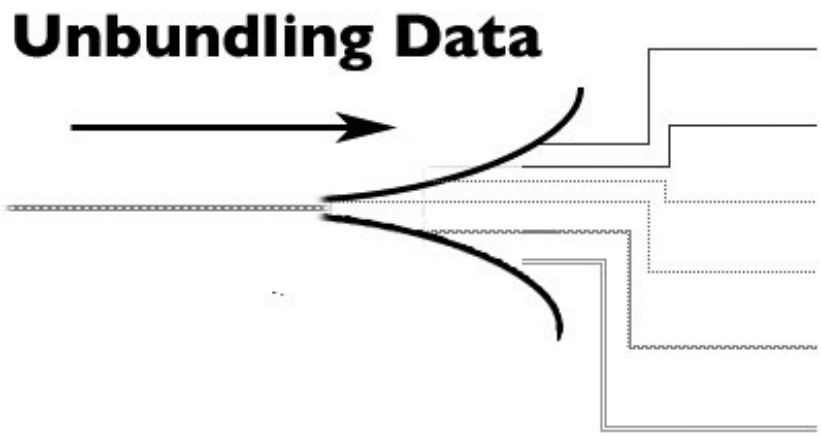


Figure 7.43. A conceptual illustration showing the unbundling of data elements from a cluster

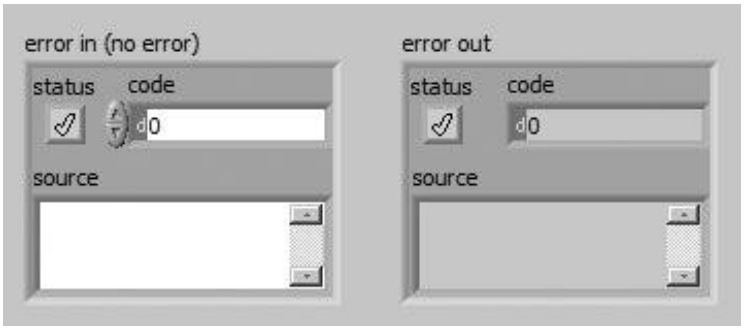


You can access cluster elements by *unbundling* them all at once or by indexing one at a time, depending on the function you choose; each method has its place (see Figure 7.43). Think of unbundling a cluster as unwrapping a telephone cable and having access to the different-colored wires. Unlike arrays, which can change size dynamically, clusters have a fixed size, or a fixed number of wires in them.

You can connect cluster terminals with a wire *only* if they have exactly the same type; in other words, both clusters must have the same number of elements, and corresponding elements must match in both data type and order. The principle of polymorphism applies to clusters as well as arrays, as long as the data types match.

You will often see clusters used in error handling. Figure 7.44 shows the error clusters, Error In.ctl and Error Out.ctl, used by LabVIEW VIs to pass a record of errors among multiple VIs in a block diagram (for example, many of the data acquisition and file I/O VIs have error clusters built into them). These error clusters are so frequently used that they appear in the Modern>>Array, Matrix & Cluster subpalette of the Controls palette for easy access.

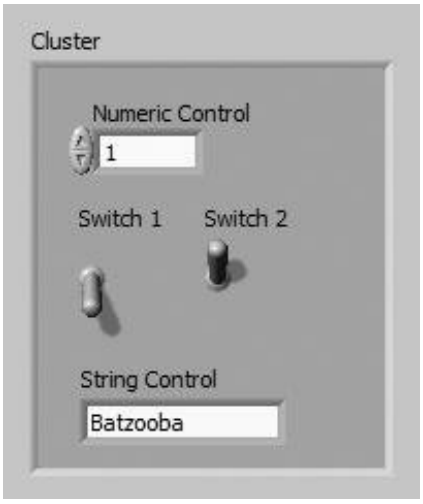
Figure 7.44. Front panel containing Error In and Error Out clusters, which are used for error handling



Creating Cluster Controls and Indicators

Create a cluster by placing a Cluster shell (Modern>>Array, Matrix & Cluster subpalette of the Controls palette) on the front panel. You can then place any front panel objects inside the cluster. Like arrays, you can deposit objects directly inside when you pull them off of the Controls palette, or you can drag an existing object into a cluster. *Objects inside a cluster must be all controls or all indicators*. You cannot combine both controls and indicators inside the same cluster, because the cluster itself must be one or the other. The cluster will be a control or indicator based on the status of the first object you place inside. Resize the cluster with the Positioning tool if necessary. Figure 7.45 shows a cluster with four controls.

Figure 7.45. Front panel with a cluster containing four controls



You can create block diagram cluster constants in a similar two-step manner.

If you want your cluster to conform exactly to the size of the objects inside it, pop up on the *border* (NOT inside the cluster) and choose an option in the Autosizing menu.

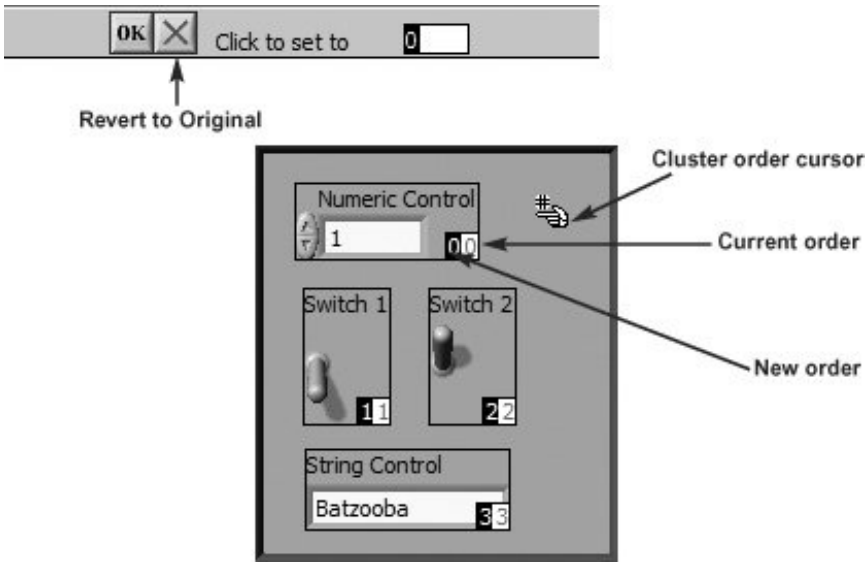
Cluster Order

Cluster elements have a logical order unrelated to their position within the shell. The first object placed in the cluster is element zero, the second is element one, and so on. If you delete an element, the order adjusts automatically. *You must keep track of your cluster order if you want to connect your cluster to another clusterthe order and data types must be identical*. Also, if you choose to unbundle the cluster all at once, you'll want to know which value corresponds to which output on the cluster function (more about unbundling will be discussed later in this chapter in the section, "Unbundling Your Clusters").

Change the order within the cluster by popping up on the cluster *border* and choosing Reorder

Controls in Cluster... from the pop-up menu. A new set of buttons appears in the Toolbar, and the cluster appearance changes, as shown in Figure 7.46.

Figure 7.46. Setting the order of cluster elements



The white boxes on the elements show their current places in the cluster order. The black boxes show the new places. Clicking on an element with the cluster order cursor sets the element's place in the cluster order to the number displayed on the Toolbar. You can type a new number into that field before you click on an object.

Revert Button

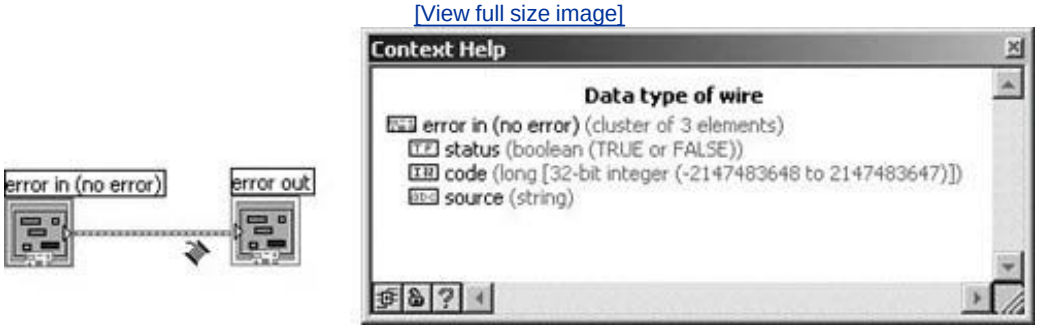
OK Button

If you don't like the changes you've made, revert to the old order by clicking on the Revert button. When you have the order the way you want, you can set it and return to your regular front panel by clicking on the OK button and exiting the cluster order edit mode.

You can quickly inspect the order of any cluster using the Context Help window. Simply move the Wiring tool over the wire, and the Context Help window will display the data type of the wirethe cluster elements are displayed in order, from top to bottom, as shown in Figure 7.47. (Show the Context Help window by selecting Help>>Show Context Help from the menu or by pressing the shortcut key <ctrl-H> [Windows], <command-H> [Mac OS X], or <meta-H> [Linux].)

Figure 7.47. The Context Help window reveals the order of cluster elements.

[View full size image]



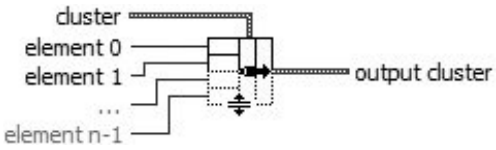
Using Clusters to Pass Data to and from SubVIs

The connector pane of a VI can have a maximum of 28 terminals. You probably don't want to pass information to all 28 terminals when calling a subVI anywaythe wiring can be very tedious and you can easily make a mistake. By bundling a number of controls or indicators into a cluster, you can use a single terminal and wire to pass several values into or out of the subVI. You can use clusters to work around the 28-terminal limit for the connector or just enjoy simplified wiring with fewer (and therefore larger and easier to see) terminals.

Bundle Function

The Bundle function (Programming>>Cluster & Variant palette) assembles individual components into a new cluster or allows you to replace elements in an existing cluster (see Figure 7.48). The function appears as the icon at the left when you place it in the diagram window. You can increase the number of inputs by dragging the resize-handles at the top or bottom edge of the function that appear as you hover the mouse over it with the Positioning tool. When you wire to each input terminal, a symbol representing the data type of the wired element appears on the empty terminal. The order of the resultant cluster will be the order of inputs to the Bundle.

Figure 7.48. Bundle



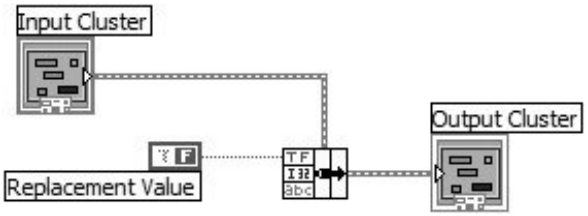
If you just want to create a new cluster, you do not need to wire an input to the center cluster input of the Bundle function. You do need to wire this input if you are replacing an element in the cluster. If you do wire this input, the Bundle will automatically adjust to match the input cluster.

Be careful with this functionif you add an element to your cluster without adding an element to the block diagram Bundle to match (or *vice versa*), your program will break!

Replacing a Cluster Element

If you want to *replace* an element in a cluster, wire the cluster to the middle terminal of the Bundle function (symbols for data types inside the cluster will appear in the Bundle inputs) and wire the new value(s) of the element(s) you want to replace to the corresponding inputs. You need to wire only to those terminals whose values you want to change (see Figure 7.49).

Figure 7.49. Replacing a cluster element using the Bundle function



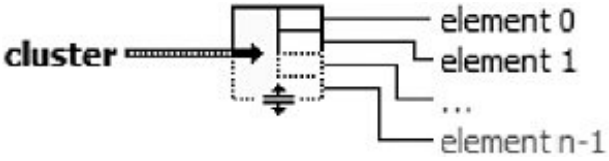
If you hover the wiring tool over the input terminals of the bundle node, the tip strip will show you the element name, if any.

Unbundling Your Clusters

Unbundle Function

The Unbundle function (Programming>>Cluster & Variant palette) splits a cluster into each of its individual components (see Figure 7.50). *The output components are arranged from top to bottom in the same order they have in the cluster*. If they have the same data type, the elements' order in the cluster is the only way you can distinguish between them. The function appears as the icon at left when you place it in the diagram window. When you wire an input cluster to an Unbundle, the Unbundle function automatically resizes to contain the same number of outputs as there are elements in the input cluster and the previously blank output terminals assume the symbols of the data types in the cluster.

Figure 7.50. Unbundle



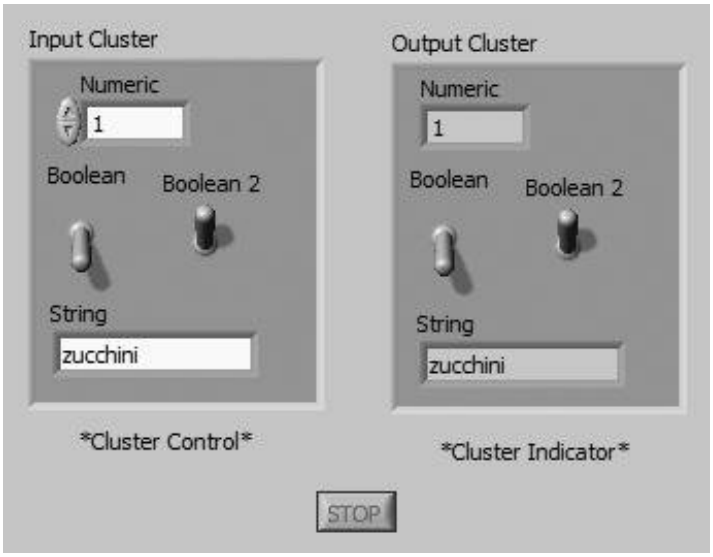
Knowing the cluster order is essential when accessing cluster data with Bundle and Unbundle. For example, if you have two Booleans in the same cluster, it would be easy to mistakenly access `Switch 2` instead of `Switch 1` because they are referenced in the Unbundle function by order, NOT by name. Your VI will wire with no error, but your results will be incorrect.

LabVIEW does have a way to bundle and unbundle clusters using element names; we'll talk about it in a minute.

Activity 7-4: Cluster Practice

In this activity, you will build a VI to teach you how to work with clusters. You will create a cluster, unbundle it, and then re-bundle it and display the values in another cluster (see Figure 7.51).

Figure 7.51. Front panel of the VI you will create during this activity



1. Open a new panel and place a Cluster shell (Modern>>Array, Matrix & Cluster palette) on it. Label it `Input Cluster`. Enlarge the shell (make sure to grab the cluster border or nothing will happen).

2. Place a digital control, two Boolean switches, and a string control inside the `Input Cluster` shell.

3. Now create `Output Cluster` by cloning `Input Cluster`. Then pop up on an object in the cluster (or on the cluster border) and select Change to Indicator. Also change the new cluster's label. (Remember, <control>-drag under Windows, <option>-drag on the Mac, <meta>-drag on Sun, and <alt>-drag on Linux to clone an object. This technique ensures correct cluster order as well as being very efficient.)

   You could also create `Output Cluster` the same way you created `Input Cluster`, using indicators instead of controls (make sure to put elements into the cluster in the same order).

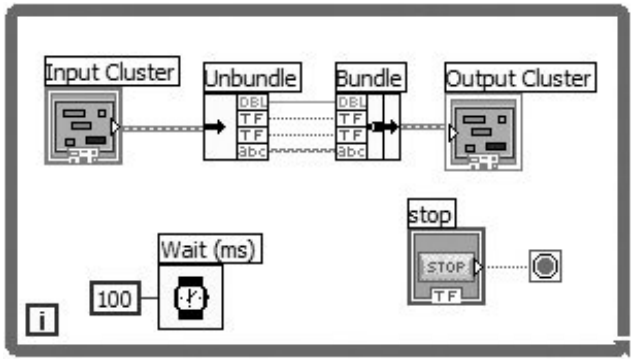
4. Verify that `Input Cluster` and `Output Cluster` have the same cluster order by popping up on each cluster border and choosing Reorder Controls in Cluster.... If the cluster orders are different, change one of them so both are the same.

5. Finally, place a Rectangular Stop Button (Boolean palette) on the front panel. Note that this button is FALSE by default. Do not change its state.

6. Build the block diagram shown in Figure 7.52. Notice that even though each cluster contains four objects, you see only one terminal per cluster on the block diagram. Make sure you change (if necessary) the conditional terminal of the While Loop so that it is set to Stop if True (you can do this by popping up on the conditional terminal).

Stop if True

Figure 7.52. Block diagram of the VI you will create during this activity



Unbundle Function

Unbundle function (Cluster palette). This function will break out the cluster so you can access individual elements. It will resize so that it has four outputs, and the data-type labels will appear on the Unbundle function after you wire `Input Cluster` to it.

Bundle Function

Bundle function (Cluster palette). This function reassembles your cluster. Resize it so that it contains four inputs.

You can also access Bundle and Unbundle by popping up on a cluster terminal or wire and choosing the function you want from the Cluster and Variant Palette menu.

7. Return to the front panel and run the VI. Enter different values for the control cluster and watch how the indicator cluster echoes the values. Press the `STOP` button to halt execution.

   You might have noticed that you could really just wire from the `Input Cluster` terminal to the `Output Cluster` terminal and the VI would do the same thing, but we wanted you to practice with Bundle and Unbundle.

8. Save the VI as Cluster Exercise.vi in your `MYWORK` directory or VI library.

9. Change the wiring so that you swap the connections between the two Boolean wires that go from between the Unbundle and the Bundle (the wires will be crossed). Run the VI and see the confusion that results. This is one reason why experienced LabVIEW users usually use the Bundle by Name and Unbundle by Name functions what you will learn about in the next section.

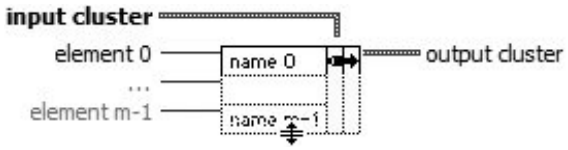
10. Close the VI without saving this confusing miswiring.

Bundling and Unbundling by Name

Sometimes you don't need to assemble or disassemble an entire clusteryou just need to operate on an element or two. You can use Bundle By Name and Unbundle By Name to do just that!

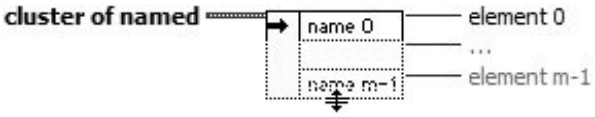
Bundle By Name, found in the Cluster palette, references elements by name instead of by position (as Bundle does). Unlike Bundle, you can access only the elements you need. However, Bundle By Name cannot create new clusters; it can only replace an element in an existing cluster. Unlike Bundle, you must always wire to Bundle By Name's middle input terminal to tell the function in which cluster to replace the element (see Figure 7.53).

Figure 7.53. Bundle By Name



Unbundle By Name, also located in the Cluster palette, returns elements whose name(s) you specify (see Figure 7.54). You don't have to worry about cluster order or correct Unbundle function size.

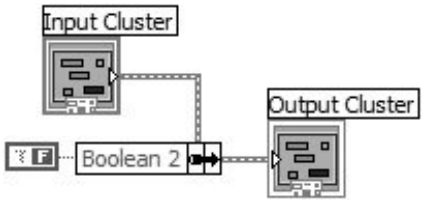
Figure 7.54. Unbundle By Name



Make sure all cluster elements have owned labels when you are using the Bundle By Name and Unbundle By Name functions. Obviously, you can't access by name if you don't have namesLabVIEW won't know what you want!

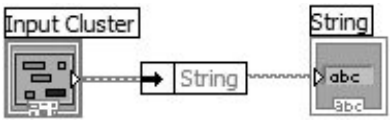
For example, if you wanted to replace the value of `Boolean 2` in the last exercise, you could use the Bundle By Name function without having to worry about cluster order or size, as shown in Figure 7.55.

Figure 7.55. Using Bundle By Name to replace the value of the **`Boolean 2`** cluster element



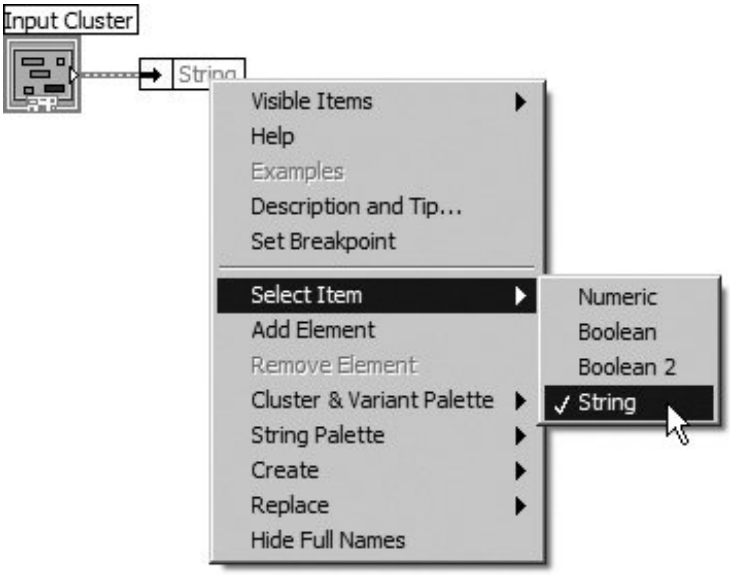
Similarly, if you only needed to access the value of `String`, you would want to use the Unbundle By Name function.

Figure 7.56. Using Unbundle By Name to get the value of the **`String`** cluster element



As soon as you wire the cluster input of Bundle By Name or Unbundle By Name, the name of the first element in the cluster appears in the name input or output. To access another element, click on the name input or output with the Operating or Labeling tool. You should see a list of the names of all labeled elements in the cluster. Select your element from this list, and the name will appear in the name terminal. You can also access this list by popping up on name and choosing Select Item>> (see Figure 7.57).

Figure 7.57. Using the pop-up menu of an output terminal of Unbundle By Name to display/modify the terminal's selected element



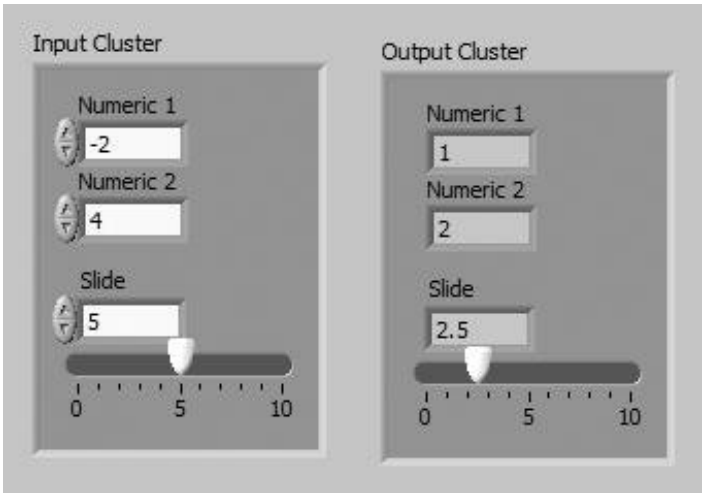
Both functions can be resized to accommodate as many elements as you need; select each component you want to access individually. As an added plus, you no longer have to worry about your program breaking if you resize your cluster. The Bundle By Name and Unbundle By Name functions do not break unless you remove an element they reference.

Activity 7-5: More Fun with Clusters

In this activity, you will build a VI that checks whether the value of the `Numeric 1` digital control in the input cluster is greater than or equal to zero. If it is less than zero, the VI will take the absolute value of all controls. If `Numeric 1` is greater than or equal to zero, the VI does not take the absolute value of any controls. Regardless of the value of `Numeric 1`, the VI multiplies all values by 0.5 and

displays the results in `Output Cluster`, demonstrating how you can use polymorphism with clusters (see Figure 7.58).

Figure 7.58. Front panel of the VI you will create during this activity



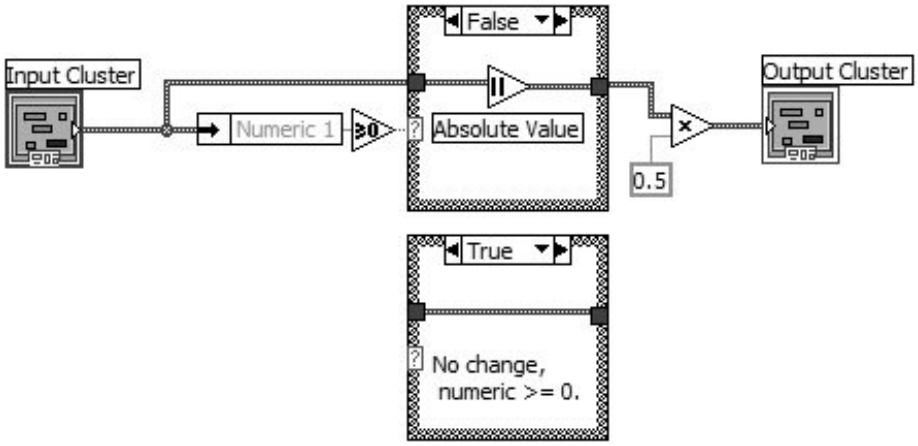
1. Open a new panel and place a Cluster shell (Modern>>Array, Matrix & Cluster palette) on it. Label it `Input Cluster`.

2. Create the `Numeric 1`, `Numeric2`, and `Slide` controls from the Programming>>Numeric palette. As you select them from the palette, click to place them inside the cluster shell. Make sure to create them in the order specified (because you will be wiring the `Input Cluster` to the `Output Cluster`) and give them labels.

3. Now create `Output Cluster` the same way using indicators (and make sure to put elements into the cluster in the same order). Or you might create `Output Cluster` by cloning `Input Cluster` and then changing its label.

4. Build the block diagram shown. Make sure to build both the TRUE and FALSE cases of the Case Structure (see Figure 7.59).

Figure 7.59. Block diagram of the VI you will create during this activity



Unbundle By Name Function

Unbundle By Name function (Programming>>Cluster & Variant palette) extracts `Numeric 1` from `Input Cluster` so you can compare it to zero. If "Numeric 1" isn't showing in the output name area, click with the Operating tool to choose it from the list of cluster elements.

Greater Or Equal To 0? Function

Greater Or Equal To 0? function (Programming>>Comparison palette). Function returns TRUE if the numeric input is zero or a positive number.

Absolute Value Function

Absolute Value (Programming>>Numeric palette) returns the input number if that number is greater than or equal to zero; returns the inverse of the input number if the input number is less than zero. In this activity, takes the absolute value of the entire cluster.

5. Run the VI. Try both positive and negative values of `Numeric 1`. Note the use of polymorphism to multiply all values in the cluster by 0.5 at the same time and to find the absolute value of the entire cluster.

6. Save the VI as Cluster Comparison.vi in your `MYWORK` directory or VI library.

PREV NEXT

Interchangeable Arrays and Clusters

Sometimes you will find it convenient to change arrays to clusters, and vice versa. This trick can be extremely useful, especially because LabVIEW includes many more functions that operate on arrays than clusters. For example, maybe you have a cluster of buttons on your front panel and you want to reverse the order of the buttons' values. Well, Reverse 1D Array would be perfect, but it only works on arrays. Have no fearyou can use the Cluster To Array function to change the cluster to an array, use Reverse 1D Array to switch the values around, and then use Array To Cluster to change back to a cluster (see Figures 7.60 and 7.61).

Figure 7.60. Array To Cluster

Figure 7.61. Cluster To Array

Cluster To Array converts a cluster of *N* elements of the same data type into an array of N elements of that type. Array index corresponds to cluster order (i.e., cluster element 0 becomes the value at array index 0). You cannot use this function on a cluster containing arrays as elements, because LabVIEW won't let you create an array of arrays. Note that all elements in the cluster must have the same data type to use this function.

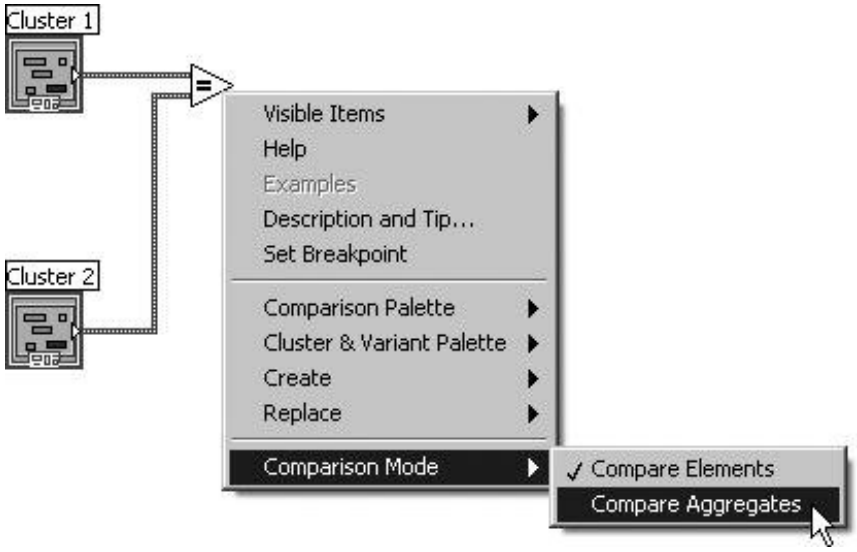
Array To Cluster converts an *N* element, 1D array into a cluster of N elements of the same data type; you must pop up on the Array To Cluster terminal and choose Cluster Size... to specify the size of the output cluster because clusters don't size automatically like arrays do. The cluster size defaults to 9; if your array is not empty but has fewer than the number of elements specified in the cluster size, LabVIEW will automatically fill in the extra cluster values with the default value for the data type of the cluster. However, if the input array has a greater number of elements than the value specified in the cluster size window, the input array will be truncated (trimmed) to the number of elements specified in the cluster size. It is also important that the output cluster size match the number of elements of the data sink to which it is wiredif it does not match, the output wire will be broken until the cluster size is adjusted appropriately.

Both functions are very handy if you want to display elements in a front panel cluster control or indicator but need to manipulate the elements by index value on the block diagram. They can be found in both the Programming>>Array and Programming>>Cluster & Variant subpalettes of the Functions palette.

Comparison Function Modes for Arrays and Clusters

Some Comparison functions have two modes for comparing arrays or clusters of data: *Compare Aggregates* mode and *Compare Elements* mode. You can select the mode you want from the Comparison Mode submenu of the comparison node's pop-up menu (shown in Figure 7.62).

Figure 7.62. Selecting the Comparison Mode of the polymorphic Add function

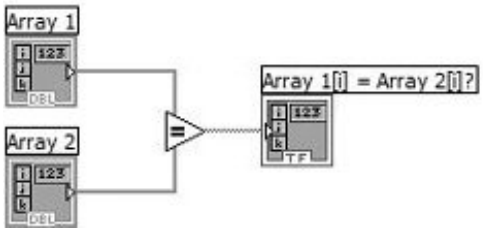


In Compare Aggregates mode, the Comparison functions return a single Boolean value that is a comparison of the entire aggregatethe returned Boolean is TRUE if and only if the comparison is TRUE for all elements. In Compare Elements mode, the Comparison functions return an array or cluster of Boolean values, which is a comparison of the individual elements, on a per-element basis. Figure 7.63 shows an example of these two different comparison modes with the Add function.

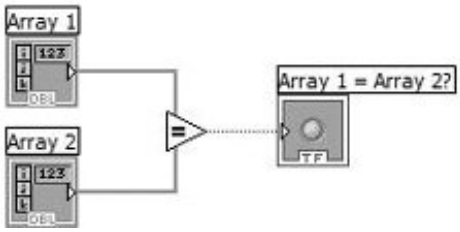
Figure 7.63. The two different modes of the polymorphic Add function: Compare Elements (left) and Compare Aggregates (right)

[View full size image]

Equal? comparison in **Compare Elements** mode:
Are Elements Equal?



Equal? comparison in **Compare Aggregates** mode:
Are Arrays (All Elelements) Equal?



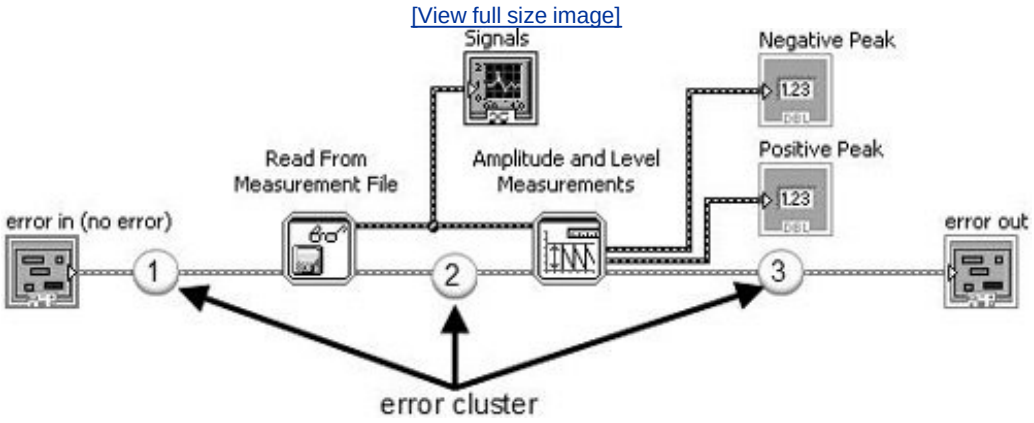
PREV NEXT

Error Clusters and Error-Handling Functions

You might have already noticed the *error cluster* datatype, which is found on the Modern>>Array, Matrix & Cluster palette (Error In and Error Out, shown in Figure 7.64) and can also be found used in the inputs and outputs of many VIs and functions. The error cluster is a special datatype in LabVIEW (a cluster consisting of a `status` Boolean, `code` I32, and `source` string) that is used for propagating information about errors that occur during the execution of LabVIEW code. Errors and error handling are very naturalthere is no reason to be afraid of errors. Rather, we should understand how errors can occur, how to communicate errors, and how to handle errors.

Figure 7.64. Error In and Error Out controls found on the Array, Matrix & Cluster palette

[View full size image]



So, what is an error? In simple terms, an error is an event where a function or subVI cannot complete a request because resources are not available or the information (arguments) passed to it are not valid.

In LabVIEW, we use dataflow to propagate information about errors inside of an error cluster datatype.

Error Cluster Datatype

The error cluster (shown in Figure 7.65) contains three elements, as follows:

1. **`status`** BooleanSignifies whether there is an error (TRUE) or no error (FALSE).

2. **code** I32Is an error code signed integer that identifies the error.

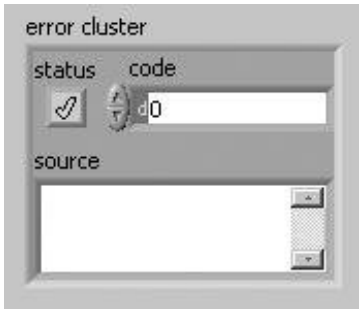
    a. Positive codes are errors.

    b. Negative codes are warnings.

    c. Zero is no error.

3. **source** stringIs a string that contains descriptive information about the source of the error. This commonly also contains a *call chain*, a list of VIs starting at the subVI (or function) where the error occurred and then listing up the call hierarchy, all the way up to the top-level VI.

Figure 7.65. Error cluster



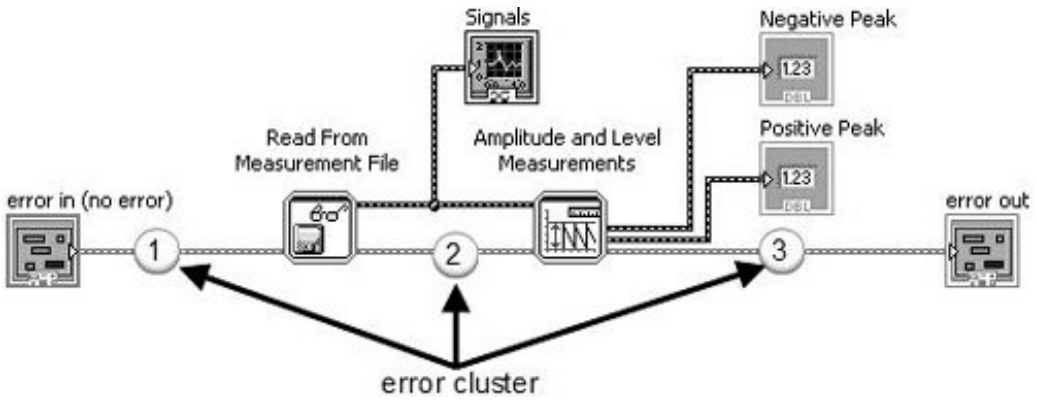
Propagating Errors: Error Dataflow

You use the *error cluster* datatype to store information about an error, and you use *dataflow* to propagate the error cluster through your block diagram. As you can see the example shown in Figure 7.66, an error cluster passes through the block diagram, connecting the following, in sequence:

1. `error in` control terminal to the Read From Measurement File subVI.
2. Read From Measurement File subVI to Amplitude and Level Measurements subVI.
3. Amplitude and Level Measurements subVI to the `error out` indicator terminal.

Figure 7.66. Block diagram showing how the error cluster is used to propagate errors using dataflow

[View full size image]



Many of LabVIEW's functions and VIs have `error in` and `error out` terminals. These are almost always found (if present on the function or VI) on the lower-left and lower-right terminals of the connector pane (respectively). Figures 7.67 and 7.68 show the inputs and outputs of Read From Measurement File and Amplitude and Level Measurements with `error in` and `error out` highlighted.

Figure 7.67. Read From Measurement File

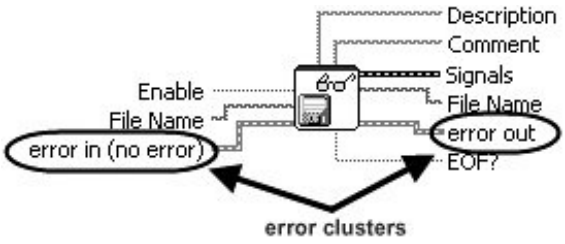
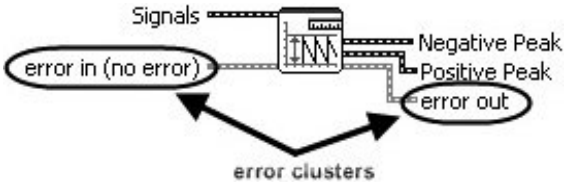


Figure 7.68. Amplitude and Level Measurements



Generating and Reacting to Errors in SubVIs

The expected behaviors of functions and VIs, with respect to generating and reacting to errors, are the following:

1. If `error in` contains an error (status = TRUE), do not do any work unless it is "clean up" work, such as

    a. closing file references.

    b. closing instrument or other communication references.

    c. putting the system back into an idle/safe state (powering off motors, etc.).

2. If an error occurs inside of a function or VI, the function should pass out the error information via its `error out` indicator terminal unless it was passed in an error via its `error in` input terminal. In this case, just pass the incoming error from `error in` to `error out`, unchanged.

Error Case Structure

A subVI can fulfill expected behavior #1, by "wrapping" all of its functional code (its "work") inside of an *Error Case Structure*, which is simply *a case structure that has an error cluster wired to its case selector terminal*. As you can see in Figure 7.69, the Case Structure allows the error cluster datatype to be wired to its case selector terminal. When wired in this way, the Case Structure frame names change to "No Error" and "Error." At run-time, if the error cluster does not contain an error, then the "No Error" frame will execute. Conversely, if the error cluster does contain an error, then the "Error" frame will execute.

Figure 7.69. An "Error Case Structure" containing an "Error" and "No Error" cases

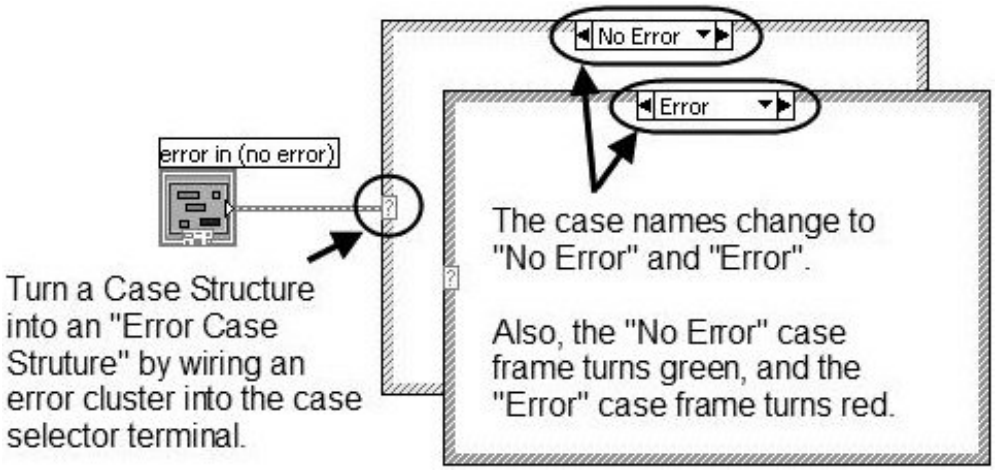
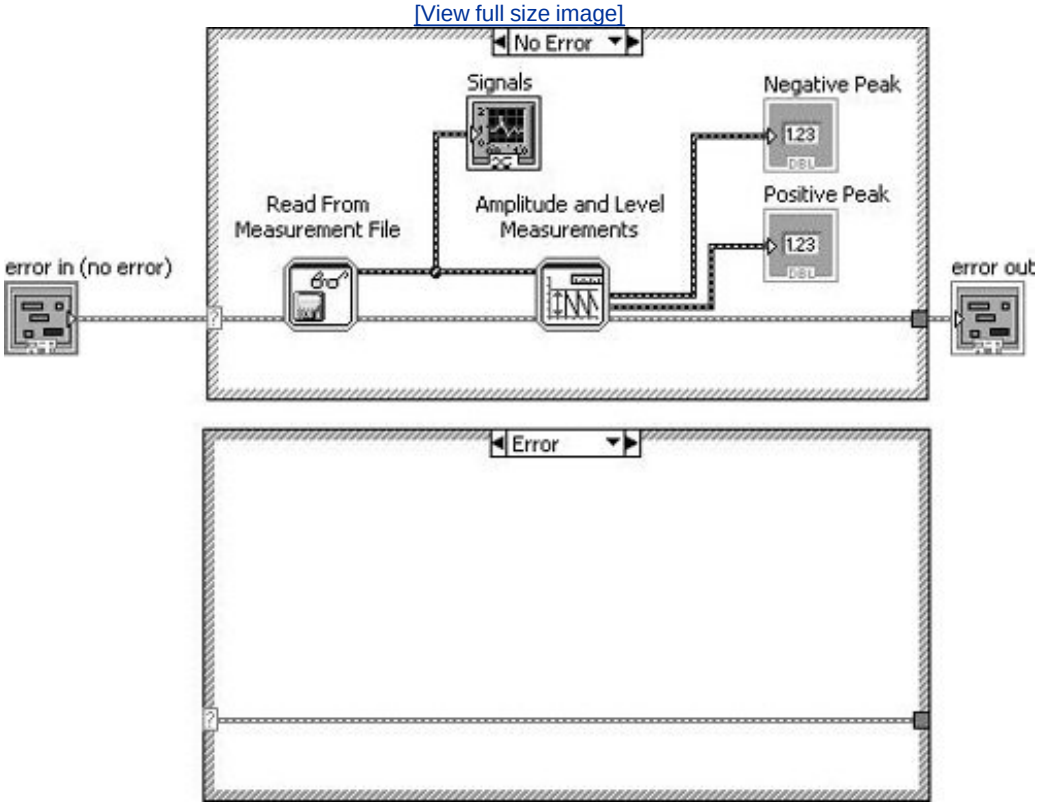


Figure 7.70 shows how we can use a case structure to conditionally execute our functional code only when there is no "upstream error" (using the dataflow vernacular).

Figure 7.70. An "Error Case Structure" used to execute code only when there is no "upstream error"

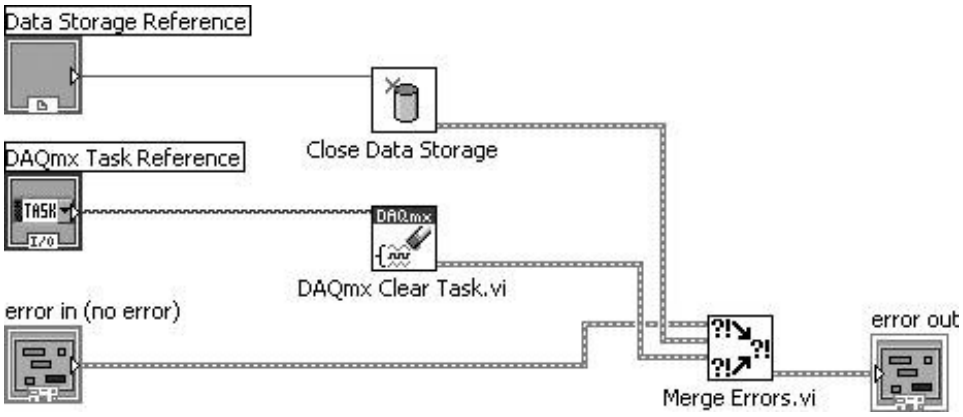
[View full size image]



Merge Errors

But, what about the situation where you want to do some clean-up work, even if an upstream error has occurred? In this case, you should not wrap your functional code in an *Error Case Structure*, but rather, merge the error cluster flowing out of your work with the upstream error cluster using Merge Errors.vi, as shown in Figure 7.71.

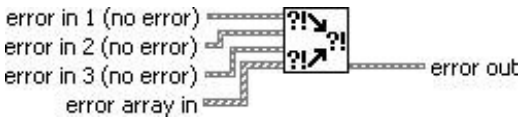
Figure 7.71. Using Merge Errors.vi to combine several error clusters into a single error cluster



An error cluster can contain information about one, and only one, error. (This is a limitation in the data structure that carries a long legacy.) Merge Errors.vi merges several errors cluster inputs into a single error output. However, if there are errors in more than one input, it must choose which of these errors will take top priority and be passed to `error out`*. It uses a top-to-bottom priority scheme, giving* `error in 1` *top priority, then* `error in 2`*, and so on.*

In the example shown in Figure 7.72, it is critical that the upstream error be wired to `error in 1` *(the top-most error input terminal) to ensure that the upstream error takes priority over any errors that occur inside our subVI.*

Figure 7.72. Merge Errors.vi



(Programming>>Dialog & User Interface palette). Merges error I/O clusters from different functions. This VI first looks for errors among error in 1, error in 2, and error in 3; then error array in and reports the first error found. If the VI finds no errors, it looks for warnings and returns the first warning found. If the VI finds no warnings, it returns no error.

Use merge errors to combine the error clusters of parallel tasks, or tasks that must each execute regardless of upstream errors.

Handling Errors in SubVIs

It is a good practice for a subVI to make a reasonable attempt to fulfill its contract (its stated job, per the software design requirements) in every way it can before passing an error up.

For example, imagine you are writing a routine that initializes an XY motion stage. (This might actually be the case, for some of our readers.) In order to find the home position (X = 0, Y = 0), the stage is moved at a low velocity toward a position sensor that outputs a digital signal of 1 (TRUE) when the stage reaches the home position. However, because there is a lot of electrical noise coming from the stage motors, the position sensor sometimes fails to report when the stage is at the home position (it sometimes jumps to 0, momentarily). This causes the initialization routine to fail at a rate of about one (1) out of twenty (20) tries, or 5% of the time. When it does fail, it always outputs the same error code. The noisy home position sensor signal has no other effect on the system, except for the occasional error during the homing operation.

Knowing what you know about the rate of failure and the type of failure, why not adapt your homing routine to retry (up to a certain number of times) before actually outputting an error message to the calling VI? This would make your code *robust:* fault tolerant and able to get the job done in spite of minor errors. Wouldn't your end users be much happier if you made this change? Think of all those times they've started an experiment and came back to the lab an hour later expecting to find their experiment completed but, instead, they find that the system has generated that blankity-blank error while trying to home the stage... again!

OK, you agree that handling errors in subVIs is a great idea. Because the error cluster is just that, a cluster, you can use the cluster functions such as Unbundle By Name and Bundle By Name to access and modify the error data. For example, you can unbundle the error code.

Generating Errors in SubVIs

When you call a subVI or function and it generates an error, you can either try again (or perhaps try something else) or you can give up and propagate error (passing it downstream or up to the calling VI). But what happens when you want to generate a new error, perhaps because of an invalid input passed down from the calling VI? For this situation, you should use Error Cluster From Error Code.vi to generate a new error (see Figure 7.73).

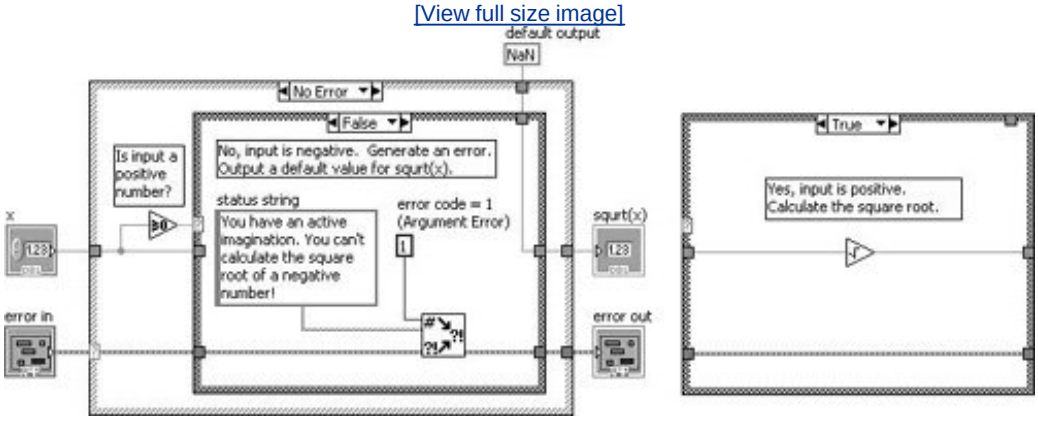
Figure 7.73. Error Cluster From Error Code.vi

Error Cluster From Error Code.vi (Programming>>Dialog & User Interface palette) converts an error or warning code to an error cluster. This VI is useful when you receive a return value from a DLL call or when you return user-defined error codes.

For example, Figure 7.74 shows how we might generate an error in a subVI when an invalid argument is passed into the subVI (the TRUE case of the inner Case Structure is shown only for illustration purposes). In this example (which is located on the CD at `EVERYONE/CH07/Calculate Square Root with Error.vi`), we are calculating the square root of a number `x`. If `x` is negative, we will output an error (error code 1, which signifies an *argument error*) and default output data.

Figure 7.74. Calculate Square Root with Error.vi block diagram

[View full size image]



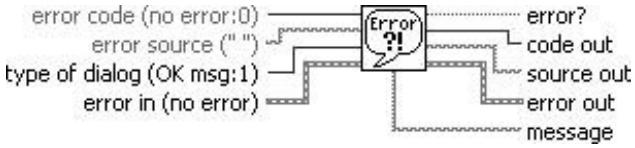
Giving Up: Displaying Error Messages to the User

If error conditions cannot be handled by subVIs or in your top-level application, you can "give up" (but, please don't accept defeat too easily) and display an error message to the user. This is the "last resort" of error handling. Figure 7.75 shows an error being passed to Simple Error Handler.vi (found on the Programming>>Dialog & User Interface palette) to display a dialog containing the error information. In this case, we passed a negative number to the subVI shown in Figure 7.74, which is an invalid input.

Figure 7.75. Calling Calculate Square Root with Error.vi as a subVI

[View full size image]

Figure 7.76. Simple Error Handler.vi



Simple Error Handler.vi (Programming>>Dialog & User Interface palette) indicates whether an error occurred. If an error occurred, this VI returns a description of the error and optionally displays a dialog box. This VI calls the General Error Handler VI and has the same basic functionality as General Error Handler but with fewer options.

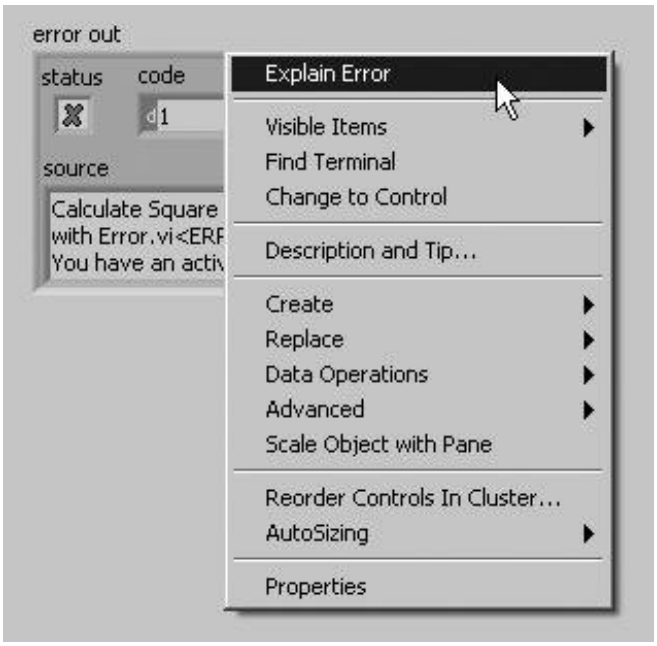
Extra Tips for Error Handling

Use the following tips for successful error handling.

Use the Explain Error Dialog

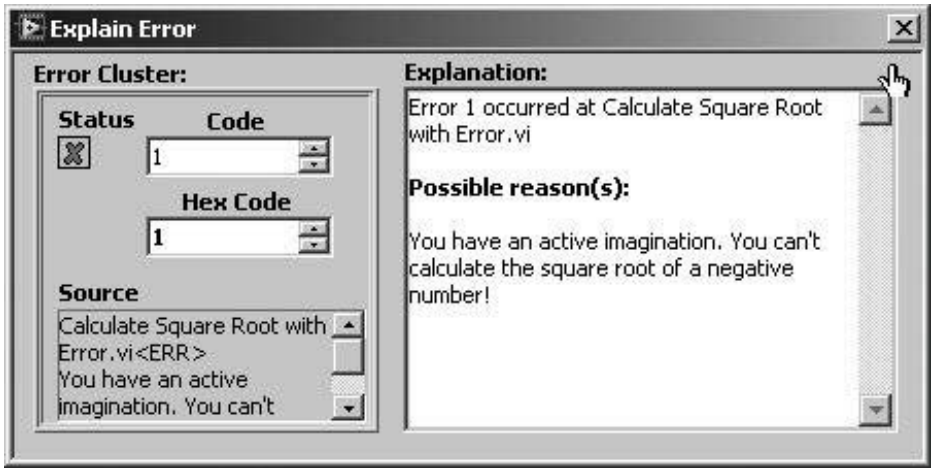
When an error cluster control or indicator contains an error or warning, you can learn more about an error by selecting Explain Error from the cluster's pop-up menu, as shown in Figure 7.77.

Figure 7.77. Selecting Explain Error from the pop-up menu of an error cluster to open the Explain Error dialog (shown in Figure 7.78)



This will display the Explain Error dialog, as shown in Figure 7.78.

Figure 7.78. Explain Error dialog, which shows a detailed error explanation



Go with the Dataflow: Use Error In and Error Out Terminals

When you add `error in` and `error out` I/O terminals to your VIs, you allow calling VIs the opportunity to *chain* the error cluster wire to create dataflow dependencies between subVIs. Also, you

are enabling applications to perform error handling, which is a good programming practice. Even if your subVI might not ever generate an error itself, put error I/O terminals on the front panel, and put an *Error Case Structure* on the block diagram (as you just learned) to allow errors to propagate through the software.

Make sure to wire error in to the lower-left terminal and error out to the lower-right terminal of the VI connector panethis is a best practice.

Define Your Own Errors: User-Defined Error Codes

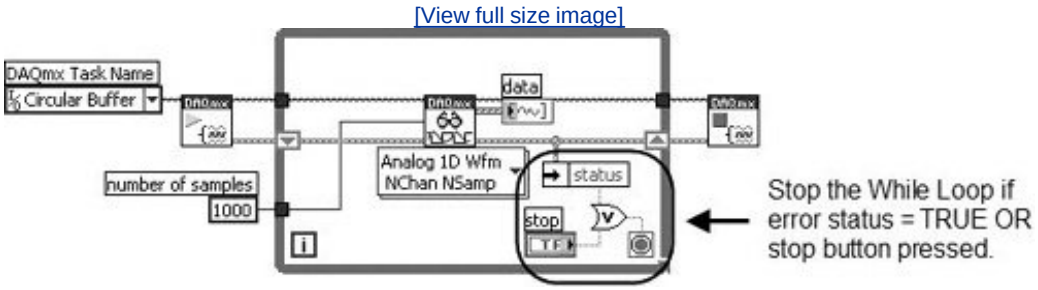
For large applications, you might want to explore user-defined error codes, which are specific to your application. The LabVIEW Help documentation describes the process for defining and using this feature.

Don't Get Stuck in a Loop: Test Error Status in Loops

It is almost always a good idea to check for errors inside of loops, so that you can exit the loop if an error occurs. Figure 7.79 shows how this is done. Note that the loop will stop if either the `stop` button is pressed *or* an error occurs.

Figure 7.79. Checking for errors in a While Loop to exit the loop when there is an error

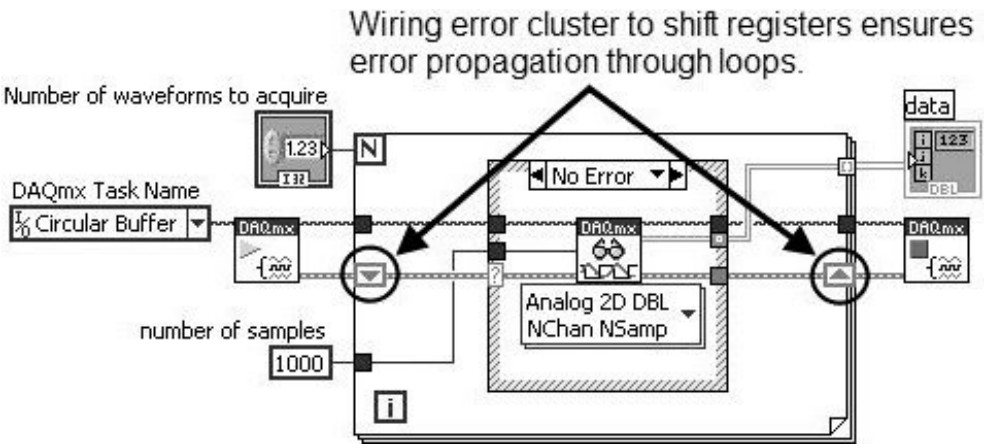
[View full size image]



Use Shift Registers to Pass Errors Through Loops

Always use shift registers (not tunnels) for passing error clusters through the wall of a loop. This is especially important for a For Loop, as shown in Figure 7.80. Remember, if a For Loop executes zero times, data still passes from the input shift register to the output shift register. Wiring error clusters to shift registers on a For Loop ensures that errors propagate through the For Loop if it executes zero times.

Figure 7.80. Using shift registers to pass errors through loops


Wiring error cluster to shift registers ensures error propagation through loops.
Number of waveforms to acquire
1.23
I32
N
data
123
DBL
DAQmx Task Name
Circular Buffer
DAQmx
No Error
Analog 2D DBL
NChan NSamp
number of samples
1000
i

PREV NEXT

Wrap It Up!

An *array* is a collection of ordered data elements of the same type. In LabVIEW, arrays can be of any data type, except chart, graph, or another array. You must create an array using a two-step process: first, place an array shell (Modern>>Array, Matrix & Cluster subpalette of the Controls palette) in the window, and then add the desired control, indicator, or constant to the shell.

LabVIEW offers many functions to help you manipulate arrays, such as Build Array and Index Array, in the Programming>>Array subpalette of the Functions palette. Most often, you will use these array functions to work with only 1D arrays; however, these functions are smart and will work similarly with multidimensional arrays (although sometimes you need to resize them first).

Both the For Loop and the While Loop can accumulate arrays at their borders using *auto-indexing*, a useful feature for creating and processing arrays. Remember that by default, LabVIEW enables indexing in For Loops and disables indexing for While Loops.

Polymorphism is a fancy name for the ability of a function to adjust to inputs of different-sized data. We talked about polymorphic capabilities of arithmetic functions; however, many other functions are also polymorphic.

Clusters also group data, but unlike arrays, they will accept data of different types. You must create them in a two-step process: first, place a cluster shell (Modern>>Array, Matrix & Cluster subpalette of the Controls palette) on the front panel or block diagram (for an array constant), and then add the desired controls, indicators, or constants to the shell. Keep in mind that objects inside a cluster must be all controls or all indicators. You cannot combine both controls and indicators within one cluster.

Clusters are useful for reducing the number of wires or terminals associated with a VI. For example, if a VI has many front panel controls and indicators that you need to associate with terminals, it is easier to group them as a cluster and have only one terminal.

The Unbundle function (Programming>>Cluster & Variant palette) splits a cluster into each of its individual components. Unbundle By Name works similarly to Unbundle, but accesses elements by their label. You can access as many or as few elements as you like using Unbundle By Name, while you have access to the whole cluster using Unbundle (and have to worry about cluster order).

The Bundle function (Programming>>Cluster palette) assembles individual components into a single cluster or replaces an element in a cluster. Bundle By Name can't assemble clusters, but it can replace individual elements in a cluster without accessing the entire cluster. In addition, with Bundle By Name, you don't have to worry about cluster order or correct Bundle function size. Just make sure all cluster elements have names when using Bundle By Name and Unbundle By Name!

The *error cluster* is a special datatype in LabVIEW (a cluster consisting of a `status` Boolean, `code` I32, and `source` string) that is used for propagating information about errors that occur during the execution of LabVIEW code. Many of LabVIEW's functions and VIs have `error in` and `error out` terminals. These are almost always found (if present on the function or VI) on the lower-left and lower-right terminals of the connector pane (respectively). Chain the VIs with error in and error out together for enforcing dataflow and ensuring that errors propagate through your application.

When building subVIs, it is important to conform to the standard expectations with respect to error handling and propagation. Put functional code inside an Error Case Structure to not execute that code when an error flows into your VI. Merge errors coming from VIs executing in parallel. Don't pop up error dialogs from subVIs. Use error dialogs in your main application only when you cannot handle them in a suitable fashion.

PREV NEXT

Additional Activities

Activity 7-6: Reversing the Order Challenge

Build a VI that reverses the order of an array containing 100 random numbers. For example, array[0] becomes array[99], array[1] becomes array[98], and so on. Name the VI Reverse Random Array.vi.

Activity 7-7: Taking a Subset

Build a VI that generates an array containing 100 random numbers and displays a portion of the array; for example, from index 10 to index 50. Name the VI Subset Random Array.vi.

Use the Array Subset function (Programming>>Array palette) to extract the portion of the array.

Activity 7-8: Dice! Challenge

Build a VI that simulates the roll of a die (possible values 16) and keeps track of the number of times that the die rolls each value. Your input is the number of times to roll the die, and the outputs include (for each possible value) the number of times the die fell on that value. Name the VI Die Roller.vi.

You will need to use a shift register in order to keep track of values from one iteration of a loop to the next.

Activity 7-9: Multiplying Array Elements

Build a VI that takes an input ID array, and then multiplies pairs of elements together (starting with elements 0 and 1) and outputs the resulting array. For example, the input array with values 1, 23, 10, 5, 7, 11 will result in the output array 23, 50, 77. Name the VI Array Pair Multiplier.vi.

PREV NEXT

8. LabVIEW's Exciting Visual Displays: Charts and Graphs

Overview

LabVIEW's charts and graphs let you display plots of data in a graphical form. Charts interactively plot data, appending new data to old so you can see the current value in the context of previous data, as the new data become available. Graphs plot pre-generated arrays of values in a more traditional fashion, without retaining previously-generated data. In this chapter, you will learn about charts and graphs, several ways to use them, and some of their special features. You will also learn about LabVIEW's special intensity charts and graphs, 3D graphs, and digital waveform graph. Finally, you will take a look at the waveform data type, a useful LabVIEW representation for time-based data.

Goals

- Understand the uses of charts and graphs
- Be able to recognize a chart's three modes: strip, scope, and sweep
- Understand mechanical action of Boolean switches
- Recognize the difference in functionalities of charts and graphs
- Know the data types accepted by charts and graphs for both single and multiple plots
- Customize the appearance of charts and graphs by changing the scales and using the palette, legend, and cursors
- Be familiar with the intensity and 3D charts and graphs to plot three dimensions of data
- Know about the digital waveform graph for displaying digital signals
- Understand the waveform data type, what its components are, and when you should use it
- Learn about dynamic data, how it relates to the waveform data type, and how to use it
- Be able to use a mixed signal graph to plot multiple types of plot data and use the multiplot cursor to view timing relationships between plots
- Use graph annotations to highlight data of interest on a plot

Key Terms

- Plot
- Waveform chart
- Plot legend
- Scale legend
- Graph palette
- Strip mode
- Scope mode
- Sweep mode
- Cursor
- Mechanical action
- Waveform graph
- XY graph
- Intensity charts and graphs
- 3D graphs
- Digital waveform graph
- Waveform data type
- Mixed signal graph
- Multi-plot cursor
- Dynamic data
- t0, dt, Y

Waveform Charts

A plot is simply a graphical display of *X* versus *Y* values. Often, *Y* values in a plot represent the data value, while *X* values represent time. The *waveform chart*, located in the Modern>>Graph subpalette of the Controls palette, is a special numeric indicator that can display one or more plots of data. Most often used inside loops, charts retain and display previously acquired data, appending new data as they become available in a continuously updating display. In a chart, the *Y* values represent the new data, and *X* values represent time (often, each *Y* value is generated in a loop iteration, so the *X* value represents the time for one loop). LabVIEW has only one kind of chart, but the chart has three different update modes for interactive data display. Figure 8.1 shows an example of a multiple-plot waveform chart.

Figure 8.1. Waveform Chart with multiple plots

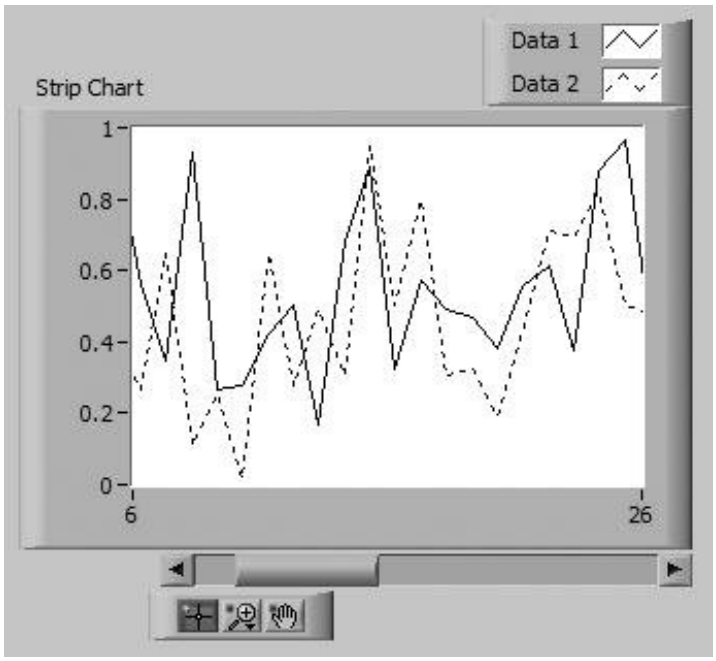


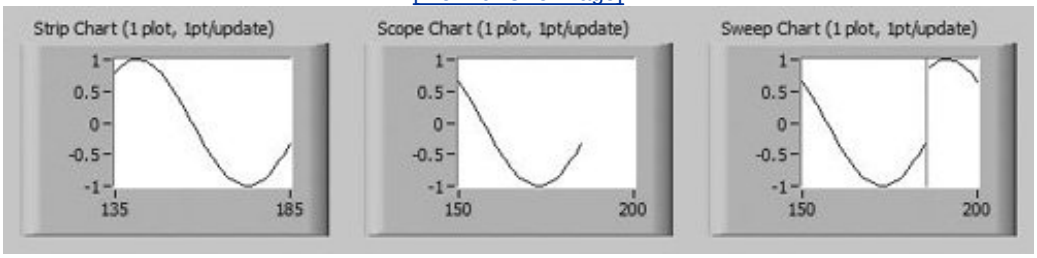
Chart Update Modes

The waveform chart has three update modes*strip chart* mode, *scope chart* mode, and *sweep chart* mode, shown in Figure 8.2. The update mode can be changed by popping up on the waveform chart and choosing one of the options from the Advanced>>Update Mode>> menu. If you want to

change modes while the VI is running (and is consequently in run mode, where the menus are slightly different), select Update Mode from the chart's runtime pop-up menu.

Figure 8.2. Chart update modes

[View full size image]



The strip chart has a scrolling display similar to a paper strip chart. The scope chart and the sweep chart have retracing displays similar to an oscilloscope. On the scope chart, when the plot reaches the right border of the plotting area, the plot erases, and plotting begins again from the left border. The sweep chart acts much like the scope chart, but the display does not blank when the data reaches the right border. Instead, a moving vertical line marks the beginning of new data and moves across the display as new data are added. These distinctions are much easier to understand when you actually see the different modes in action, so don't worry if it sounds confusing now. You'll get to experiment with them in the next activity.

Because there is less overhead in retracing a plot, the scope chart and the sweep chart operate significantly faster than the strip chart.

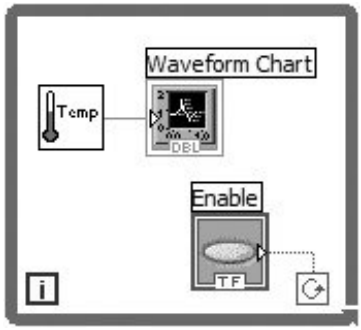
Charts assume that the X-values always represent evenly spaced points. With LabVIEW's charts, you only provide the Y value, and do not specify the X value. The X value automatically increments every time a new Y value is appended to the chart. For arbitrary X values, use a graph instead of a chart. We'll discuss graphs shortly.

The chart accepts the waveform data type (which you will learn about later in this chapter), in addition to numeric scalars and arrays. As you will learn, the waveform data type contains timing information inside of it (such as the time stamp of the first data point and the time spacing between points). The chart will use this timing information for displaying the data, which means that the initial X value and spacing between points can be different each time data are written to the chart. Once you have learned about the waveform data type, give this technique a try.

Single-Plot Charts

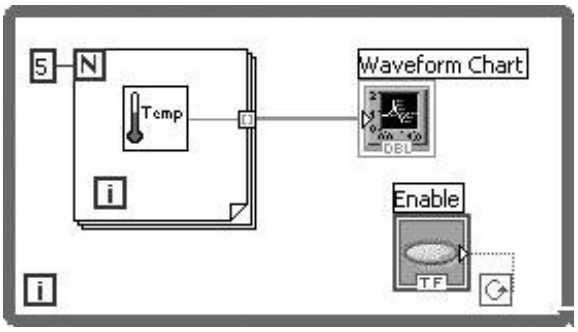
The simplest way to use a chart is to wire a scalar value to the chart's block diagram terminal, as shown in Figure 8.3. One point will be added to the displayed waveform each time the loop iterates.

Figure 8.3. Single-plot chart, updated one point at a time



You can also update a single-plot chart with multiple points at a time, as shown in Figure 8.4, by passing it an array of values.

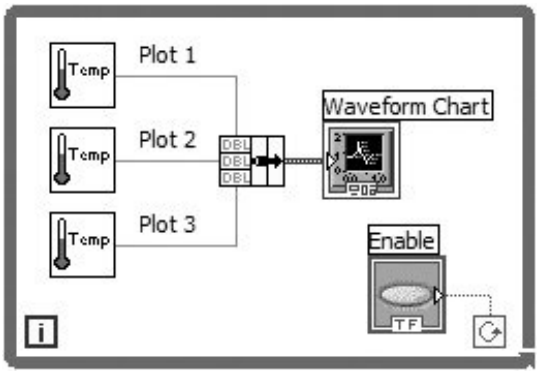
Figure 8.4. Single-plot chart, updated with multiple points at a time



Wiring a Multiple-Plot Chart

Waveform charts can also accommodate more than one plot. However, because you can't wire from multiple block diagram sources to a single chart terminal, you must first bundle the data together using the Bundle function (Programming>>Cluster & Variant palette). In Figure 8.5, the Bundle function "bundles" or groups the outputs of the three different VIs that acquire temperature into a cluster so they can be plotted on the waveform chart. Notice the change in the waveform chart terminal's appearance when it's wired to the Bundle function. To add more plots, simply increase the number of Bundle input terminals by resizing using the Positioning tool.

Figure 8.5. Bundling data points to create a multi-plot Waveform Chart



When wiring a multiple-plot chart, make sure to use a Bundle function and not a Build Array function. LabVIEW treats points in an array as belonging to a single plot and treats

points in a cluster as belonging to multiple plots. If you build an array, you will have a single plot that has three new points for each loop iteration. This behavior is useful when reading waveforms from hardware, where you are reading multiple samples for a single channel.

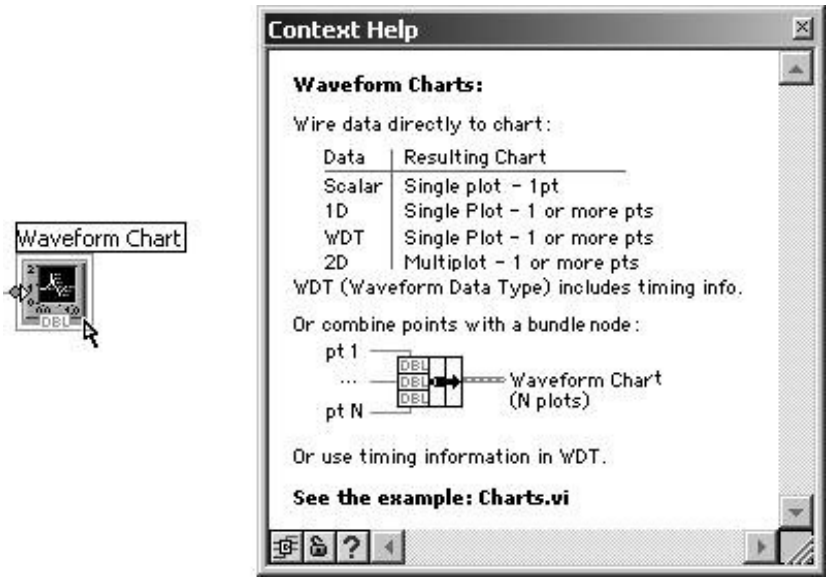
For an online example of charts, their modes, and their expected data types, open and run Charts.vi in the `EVERYONE\CH08` *directory*.

Single-Plot Versus Multi-Plot Data Types: A Trick for Remembering

Charts and graphs are polymorphic; they will accept several different data types, allowing you to create single-plots and multi-plots. However, it is often difficult to remember which data types are used for single-plot and multi-plot. Additionally, there are several different types of charts and graphs, which makes matters worse.

Fortunately, there is a way to quickly find out which data types may be used with a specific type of chart or graph. Simply hover the mouse over the chart or graph terminal, on the block diagram, and a detailed description of the plot data types will appear in the Context Help window, as shown in Figure 8.6. (Note that the *WDT* item in the third row of the table shown in the Context Help window is the waveform data type, which will soon be discussed.) The Context Help window can be made visible by selecting Help>>Show Context Help from the menu or by using the shortcut key <control-H> (in Windows), <command-H> (in Mac OS X), or <meta-H> (in Linux).

Figure 8.6. Waveform Chart terminal's context help



Show the Digital Display?

Like many other numeric indicators, charts have the option to show or hide the digital display (pop up on the chart to get the Visible Items>> option). The digital display shows the most recent value displayed by the chart.

The X Scrollbar

Charts also have an X Scrollbar that you can show or hide from the Visible Items>> pop-up submenu. You can use the scrollbar to display older data that has scrolled off the chart.

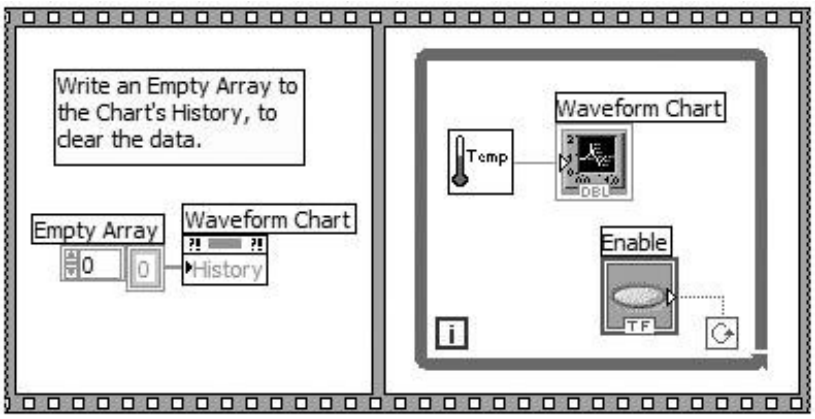
Clearing the Chart

Sometimes you will find it useful to remove all previous data from the chart display. Select Data Operations>>Clear Chart from the chart's pop-up menu to clear a chart from edit mode (remember, you are usually in edit mode if your VI is not running. To switch between modes when the VI is not running, choose Change to Run/Edit Mode from the Operate menu). If you are in run mode, Clear Chart is a pop-up menu option instead of being hidden under Data Operations.

Sometimes you will want to perform the *Clear Chart* operation programmaticallyfor example, when you first start running your VI. To do this, you will need to write an empty array to the chart control's *History Data* property, as shown in Figure 8.7. We will discuss property nodes in Chapter 13, "Advanced LabVIEW Structures and Functions," but we will give you a sneak preview now.

Figure 8.7. Writing to a Waveform Chart's History with empty data to

clear it, prior to collecting new data



1. Create a property node by popping up on the Waveform Chart terminal and selecting Create Property Node>>History Data.

2. Pop up on the History Data property node and select Change to Write, to change the History property from Read mode to Write mode.

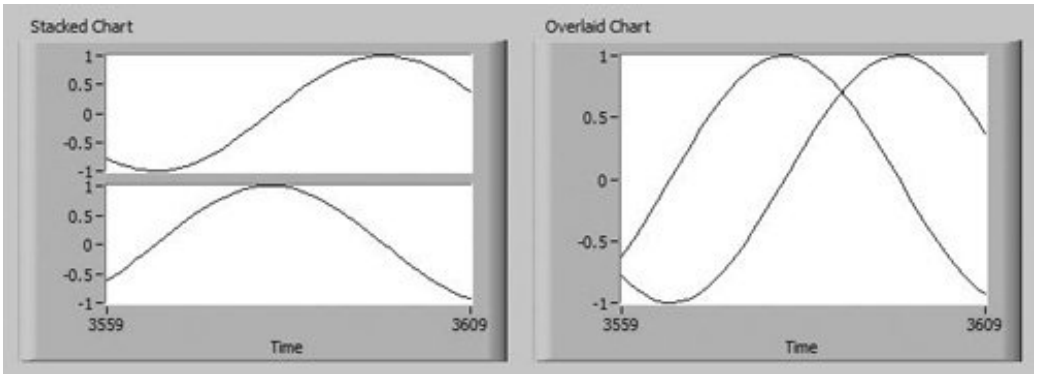
3. Pop up on the History Data property node and select Create Constant to create an empty array constant that is wired to the History property.

Stacked and Overlaid Plots

If you have a multiple-plot chart, you can choose whether to display all plots on the same Y-axis, called an *overlaid plot*; or you can give each plot its own Y scale, called a *stacked plot*. You can select Stack Plots or Overlay Plots from the chart's pop-up menu to toggle the type of display. Figure 8.8 illustrates the difference between stacked and overlaid plots.

Figure 8.8. Waveform Charts in Stack Plots mode (left) and Overlay Plots mode (right)

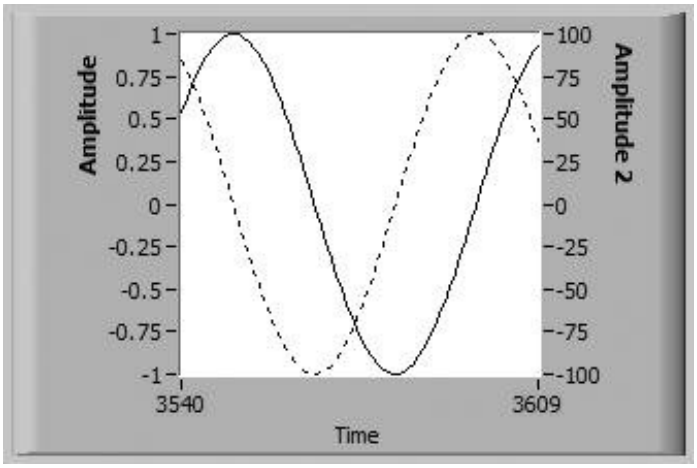
[View full size image]



Multiple Y Scales

If you have a multi-plot chart, sometimes you will want an overlaid plot that has different scales for each plot; for example, if one plot's *Y* range is from -1 to +1 and the other is -100 to +100, it would be difficult to see both of them overlaid without a separate scale for each. You can create multiple scales for the Y axis by popping up on the *Y* axis and selecting Duplicate Scale. After duplicating a scale, you may want to move it to the other side of the chart by popping up on the scale and selecting Swap Sides. To delete a *Y* scale, pop up on the scale and select Delete Scale. Figure 8.9 illustrates two *Y* scales for a multi-plot chart, each scale on one side of the chart.

Figure 8.9. Waveform Chart with two Y scales



You cannot duplicate X scales on a chartthere can be one and only one X scale on a chart. If you pop up on the X scale, you will notice that the Duplicate Scale option is grayed out. (Multiple X scales are allowed on graphs.)

You can reset the scale layout by right-clicking a graph or chart and selecting Advanced>>Reset Scale Layout from the shortcut menu. This will return the Y scale to the left of the plot area, return the X scale to the bottom of the plot area, and reset the scale markers.

Chart History Length

By default, a waveform chart can store up to 1,024 data points. If you want to store more or fewer data, select Chart History Length . . . from the pop-up menu and specify a new value of between 10 and 2,147,483,647 points (however, your actual limit may be lower, depending on the amount of RAM you have in your system). Changing the buffer size does not affect how much data are shown on the screenresize the chart to show more or fewer data at a time. However, increasing the buffer size does increase the amount of data you can scroll back through.

PREV NEXT

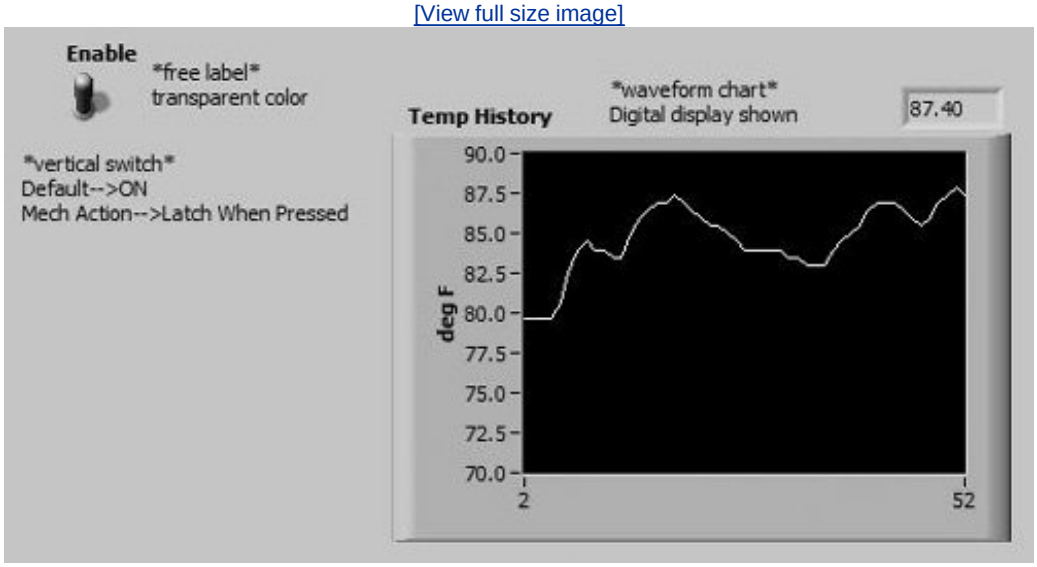
Activity 8-1: Temperature Monitor

You will build a VI to measure temperature and display it on the waveform chart. This VI will measure the temperature using the Thermometer VI you built as a subVI in a previous lesson.

1. Open a new front panel. You will recreate the panel shown in Figure 8.10 (but feel free not to type in the commentsthey're for your benefit).

Figure 8.10. Front panel of the VI you will create during this activity

[View full size image]



2. Place a vertical switch (Modern>>Boolean palette) in the front panel window. Label the switch `Enable`. You will use the switch to stop the temperature acquisition.

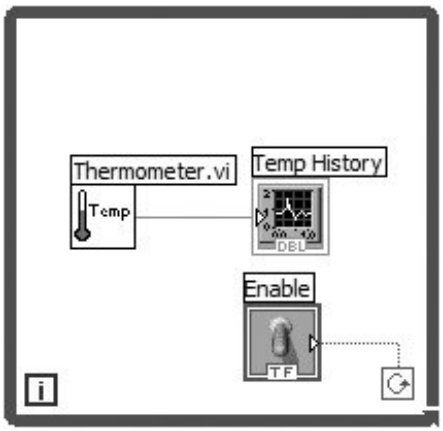
3. Place a waveform chart (Modern>>Graph palette) in the panel window. Label the waveform chart `Temp History`. The waveform chart will display the temperature in real time.

4. The waveform chart has a digital display that shows the latest value. Pop up on the waveform chart and choose Visible Items>>Digital Display from the pop-up menu.

5. Because the temperature sensor measures room temperature, rescale the waveform chart so you can see the temperature (otherwise, it will be "off the chart"). Using the Labeling tool, double-click on "10" in the waveform chart scale, type `90`, and click outside the text area. The click enters the value. You also can press <enter> to input your change to the scale. Change "10" to `70` in the same way.

6. Open the block diagram window and build the diagram shown in Figure 8.11.

Figure 8.11. Block diagram of the VI you will create during this activity



7. Place the While Loop (Programming>>Structures palette) in the block diagram window and size it appropriately. Pop up on the While Loop's Conditional Terminal and select "Continue if True" (by default, it is set to "Stop if True").

Stop if True

Continue if True

8. Place the two terminals inside the While Loop if they aren't ready there.

9. Import the Thermometer subVI.

Thermometer.vi

Thermometer.vi. This VI returns one temperature measurement from a temperature sensor

(or a simulation, depending on your setup). You should have written it in Chapter 4, "LabVIEW Foundations," and modified it in Chapter 5, "Yet More Foundations." Load it using the Select A VI . . . button on the Functions palette. It should probably be in your MYWORK directory. If you don't have it, you can use Thermometer.vi in the EVERYONE\CH05 directory or Digital Thermometer.vi, located in the <LabVIEW>\Activity folder.

10. Wire the block diagram, as shown in Figure 8.11.

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

    Remember, the While Loop is an indefinite looping structure. The subdiagram within its border will execute as long as the specified condition is TRUE, because the conditional terminal is set to Continue if True. In this example, as long as the switch is on (TRUE), the Thermometer subVI will return a new measurement and display it on the waveform chart.

12. To stop the acquisition, click on the vertical switch. This action gives the loop conditional terminal a FALSE value and the loop ends.

13. The waveform chart has a display buffer that retains a number of points after they have scrolled off the display You can show this scrollbar by popping up on the waveform chart and selecting Visible Items>>Scrollbar from the pop-up menu. You can use the Positioning tool to adjust the scrollbar's size and position.

    To scroll through the waveform chart, click on either arrow in the scrollbar.

    To clear the display buffer and reset the waveform chart, pop up on the waveform chart and choose Data Operations>>Clear Chart from the pop-up menu. If you want to clear the chart while you're in run mode, select Clear Chart from the runtime pop-up menu.

14. Make sure the switch is TRUE and run the VI again. This time, try changing the update mode of the chart. Pop up and choose Update Mode>>Scope Chart from the chart's runtime menu. Notice the difference in chart display behavior. Now choose Sweep Chart and see what happens.

Using Mechanical Action of Boolean Switches

Take a step out of this activity for a second. You've certainly noticed by now that each time you run this VI, you first must turn on the vertical Enable switch before clicking the run button, or the loop will only execute once. You can modify the *mechanical action* of a Boolean control to change its behavior and circumvent this inconvenience. LabVIEW offers six possible choices for the mechanical action of a Boolean control.

Switch When Pressed

Switch When Pressed action changes the control's value each time you click on the control with the Operating tool. This action is the default for Booleans and is similar to that of a ceiling light switch. It is not affected by how often the VI reads the control.

Switch When Released

Switch When Released action changes the control's value only when 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 mode 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

Switch Until Released action changes the control's 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 door buzzer and is not affected by how often the VI reads the control.

Latch When Pressed

Latch When Pressed action changes the control's 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. Latch When Pressed is similar in functionality to a circuit breaker and is useful when you want the VI to do something only once for each time you set the control, such as to stop a While Loop when you press a `STOP` button.

Latch When Released

Latch When Released action changes the control's 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 mode is similar to the behavior of buttons in a dialog box; the button becomes highlighted when you click on it, and latches a reading when you release the mouse button.

Latch Until Released

Latch Until Released changes the control's 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, whichever occurs last.

For example, consider a vertical switchits default value is *off* (FALSE).

15. Modify the vertical switch in your VI so that you do not need to turn the switch to TRUE each time you run the VI.

    a. Turn the vertical switch to *on* (TRUE).

    b. Pop up on the switch and choose Data Operations>>Make Current Value Default from the pop-up menu to make the on position the default value.

    c. Pop up on the switch and choose Mechanical Action>>Latch When Pressed from the pop-up menu.

16. Run the VI. Click on the vertical switch to stop the acquisition. The switch will move to the *off* position briefly, and then automatically change back to on after the While Loop conditional terminal reads one FALSE value.

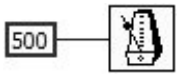
You cannot use mechanical action on a Boolean object if you will be modifying the object's value using local variables. We'll tell you why when we talk about locals in Chapter 12, "Instrument Control in LabVIEW."

Adding Timing

When you run the VI in this activity, the While Loop executes as quickly as possible. You may want to take data at certain intervals, however, such as once per second or once per minute. You can control loop timing using the Wait Until Next ms Multiple function (Programming>>Timing menu).

17. Modify the VI to take a temperature measurement about once every half-second by placing the code segment shown here into the While Loop (see Figure 8.12).

Figure 8.12. Wait Until Next ms Multiple



Wait Until Next ms Multiple Function

Wait Until Next ms Multiple function (Programming>>Timing menu) ensures that each iteration's start of execution is synchronized to a half-second (500 milliseconds). This function will wait *until* the millisecond timer's tick count reaches a multiple of the value specified. Note that this behavior is different from the Wait (ms) function, which waits for a specified *duration* of time to elapse.

18. Run the VI. Run it several more times, trying different values for the number of milliseconds.

19. Save and close the VI. Name it Temperature Monitor.vi and place it in your `MYWORK` directory or VI library. Excellent job!

PREV NEXT

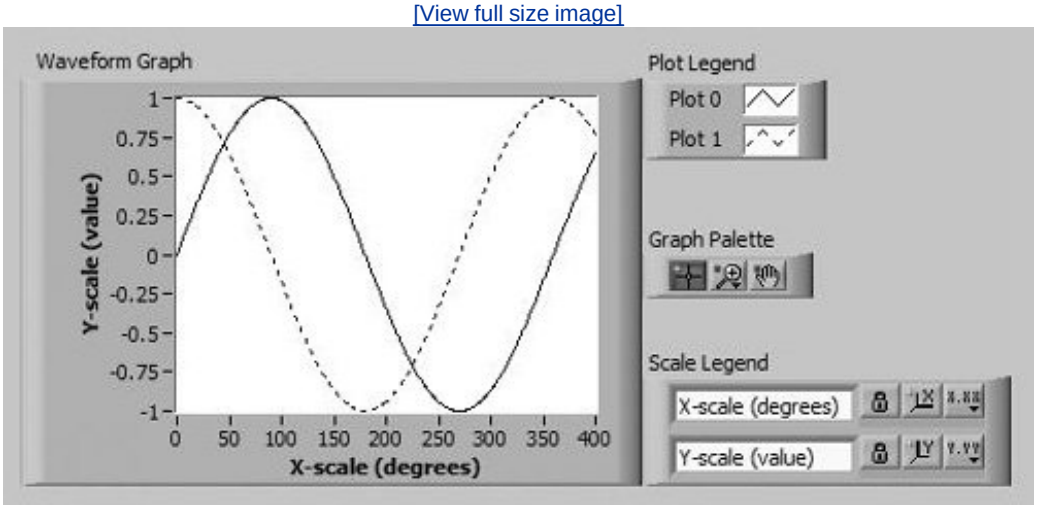
Graphs

Unlike charts, which plot data interactively, graphs plot pre-generated arrays of data all at once; they do not have the ability to append new values to previously generated data. LabVIEW provides several types of graph for greater flexibility: *waveform graphs*, *XY graphs*, *intensity graphs*, *3D graphs*, *digital waveform graphs*, and some specialized graphs (*Smith plots*, *Polar charts*, *Min-Max*, and *distribution* plots). We'll talk about waveform and XY graphs now and cover intensity and 3D graphs later, in the "3D Graphs" section of this chapter. Waveform graphs and XY graphs look identical on the front panel of your VI but have very different functionality.

An example of a graph with several graph options enabled is shown in Figure 8.13.

Figure 8.13. A Waveform Graph with several graph options enabled

[View full size image]



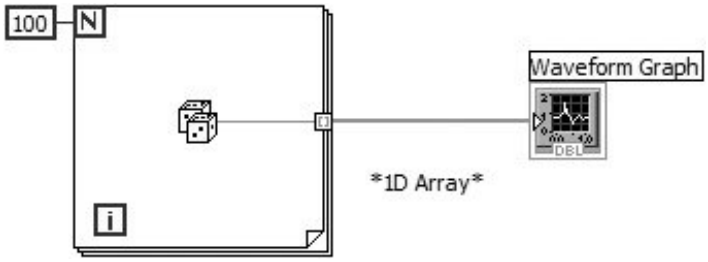
You can obtain both types of graph indicator from the Modern>>Graph subpalette of the Controls palette. The waveform graph plots only single-valued functions (only one Y value for every X) with uniformly spaced points, such as acquired time-sampled, amplitude-varying waveforms. The waveform graph is ideal for plotting arrays of data in which the points are evenly distributed.

The *XY graph* is a general-purpose, Cartesian graph, ideal for plotting data with varying timebases or data with several Y values for every X value, such as circular shapes. The two types of graph look the same but take different types of input, so you must be careful not to confuse them.

Single-Plot Waveform Graphs

For basic single-plot graphs, you can wire an array of Y values directly to a waveform graph terminal, as shown in Figure 8.14. This method assumes the initial X value is zero, and the delta X value (i.e., the increment between X values) is one. Notice that the graph terminal in the block diagram shown in Figure 8.14 appears as an array indicator (if you deselect View as Icon from its pop-up menu).

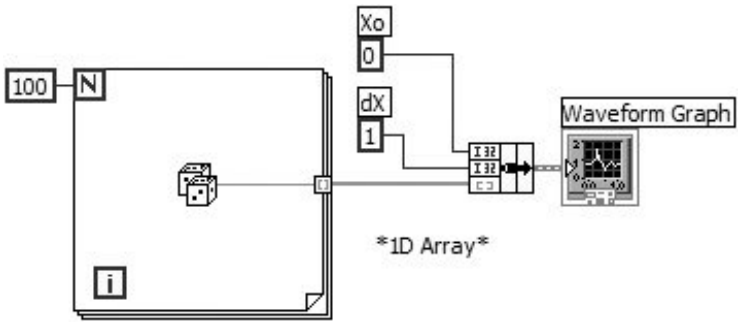
Figure 8.14. Passing a 1D array to a Waveform Graph to display a single plot with multiple points



Recall from Chapter 7, "LabVIEW's Composite Data: Arrays and Clusters," that an auto-indexing output tunnel (which is enabled by default for tunnels on For Loops) will build a 1D array from the scalar values that flow into it from inside the For Loop.

Sometimes you will want the flexibility to change the timebase for the graph. For example, you start sampling at a time other than "initial X = 0" (or $X_0 = 0$), or your samples are spaced more (or less) than one unit apart, or "delta X = 1" (also written $\Delta X = 1$). To change the timebase, bundle the X_0 value, ΔX value, and the data array into a cluster; then wire the cluster to the graph. Notice in Figure 8.15 that the graph terminal now appears as a cluster indication.

Figure 8.15. Passing a cluster with Xo, dX, and a 1D array to specify timing information of a single plot



Multiple-Plot Waveform Graphs

You can show more than one plot on a waveform graph by creating an array (or a 2D array) of the data types used in the single-plot examples (see Figure 8.16). Notice how graph terminals change appearance depending on the structure of data wired to them (array, cluster, array of clusters, etc.) and the data type (I16, DBL, etc.).

Figure 8.16. Passing a 2D array to a Waveform Graph to display two plots (plots are in rows of the 2D array and points are in columns)

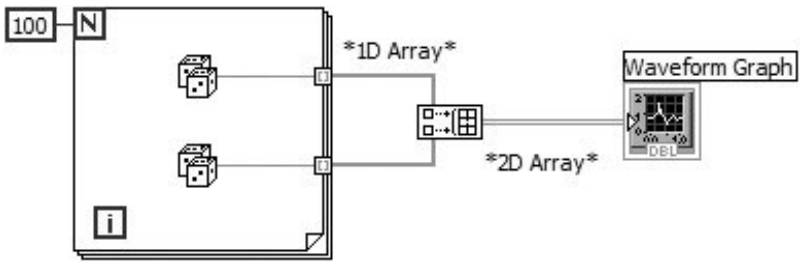
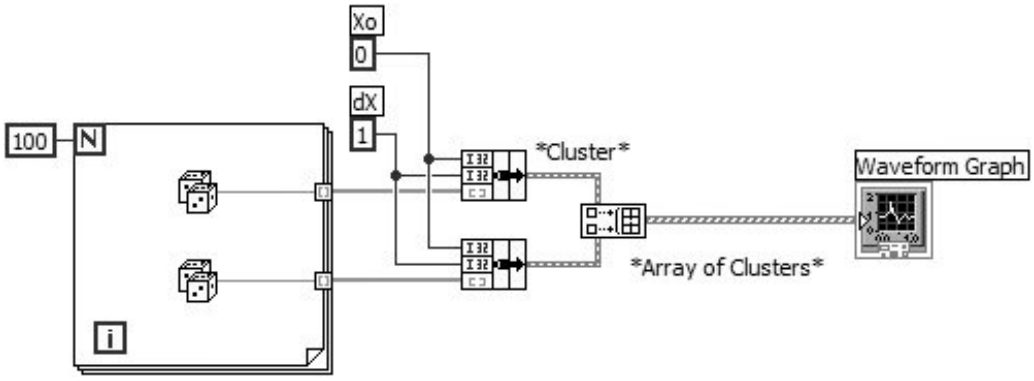


Figure 8.16 assumes that the initial X value is 0 and the delta X value is 1 for both arrays. The Build Array function creates a 2D array out of two 1D arrays. Notice that this 2D array has two rows with 100 columns per rowa 2 x 100 array. By default, graphs plot each *row* of a 2D array as a separate waveform. If your data are organized by column, you must make sure to transpose your array when you plot it! Transposing means simply switching row values with column values; for example, if you transpose an array with three rows and ten columns, you end up with an array with ten rows and three columns. LabVIEW makes it easy to do thissimply pop up on the graph and select Transpose Array (this menu option is grayed out if the graph does not have a 2D array wired to it). You can also use the Transpose 2D Array function found in the Array subpalette of the Functions menu.

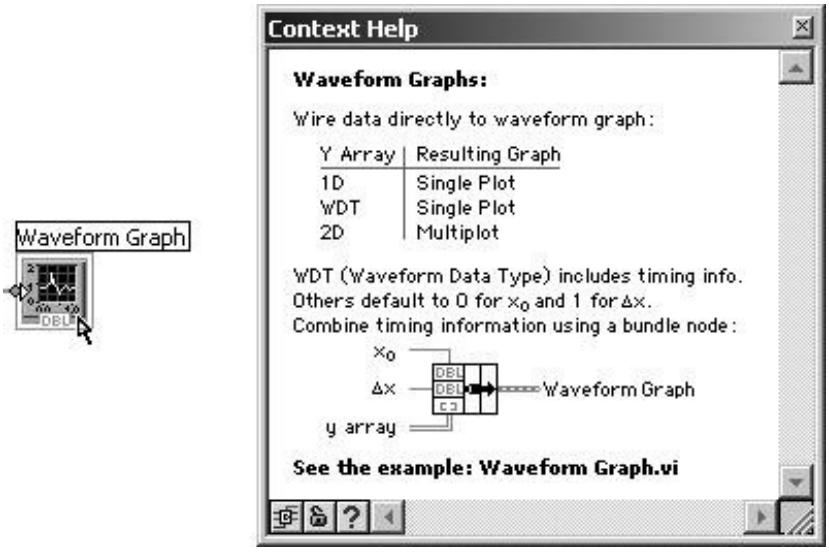
In Figure 8.17, the X_0 value and X (or delta X) value for each array is specified. These X parameters do not need to be the same for both sets of data.

Figure 8.17. Building an array of single plot (Xo, dX, and Y array) clusters to create a multi-plot



Remember to use the Context Help window to view the detailed description of the data types that may be wired to a waveform graph (as shown in Figure 8.18). This is a very useful reference. Simply hover the mouse over the graph terminal, on the block diagram, and a detailed description of the plot data types will appear in the Context Help window. The Context Help window can be made visible by selecting Help>>Show Context Help from the menu or by using the shortcut key <control-H> (in Windows), <command-H> (in Mac OS X), or <meta-H> (in Linux).

Figure 8.18. Waveform graph terminal's context help

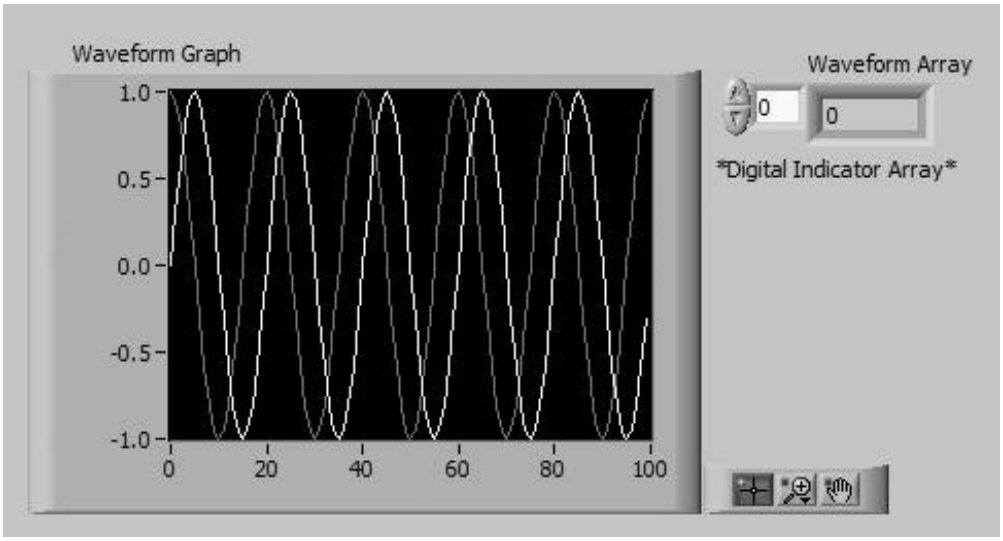


PREV NEXT

Activity 8-2: Graphing a Sine on a Waveform Graph

1. You will build a VI that generates a sine wave array and plots it in a waveform graph. You will also modify the VI to graph multiple plots (see Figure 8.19).

Figure 8.19. Front panel of the VI you will create during this activity



2. Place an array shell (Modern>>Array, Matrix & Cluster palette) in the front panel window. Label the array shell `Waveform Array`. Place a digital indicator (Numeric palette) inside the Data Object window of the array shell to display the array contents.

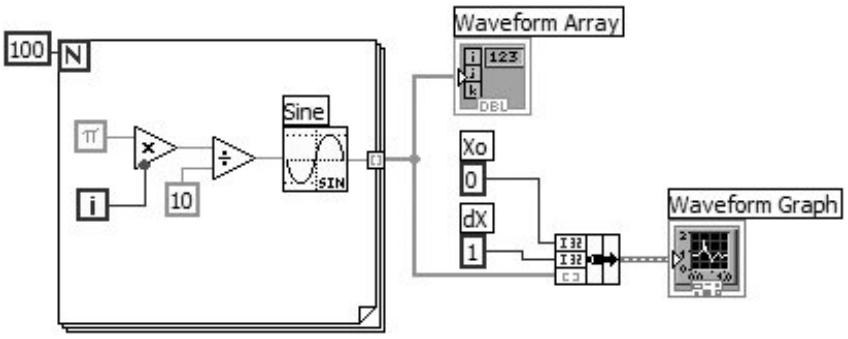
3. Place a waveform graph (Modern>>Graph palette) in the front panel window. Label the graph `Waveform Graph` and enlarge it by dragging a corner with the Positioning tool.

   Hide the legend by popping up on the graph and selecting Visible Items>>Plot Legend.

   Disable autoscaling by popping up and choosing Y Scale>>AutoScale Y. Modify the *Y* axis limits by selecting the scale limits with the Labeling tool and entering new numbers; change the *Y* axis minimum to `-1.0` and the maximum to `1.0`. We'll talk more about autoscaling in the next section.

4. Build the block diagram shown in Figure 8.20.

Figure 8.20. Block diagram of the VI you will create during this activity



Sine Function

Sine function (Mathematics>>Elementary & Special Functions>>Trigonometric Functions palette) computes `sin(x)` and returns one point of a sine wave. The VI requires a scalar index input that it expects in radians. In this exercise, the x input changes with each loop iteration and the plotted result is a sine wave.

Pi Constant

Pi constant (Programming>>Numeric>>Math & Scientific Constants palette).

Bundle Function

Bundle function (Programming>>Cluster & Variant palette) assembles the plot components into a single cluster. The components include the initial X value (0), the delta X value (1), and the Y array (waveform data). Use the Positioning tool to resize the function by dragging either of the resizing handles that appear on the top and bottom edges of the Bundle function as you hover the Positioning tool over it.

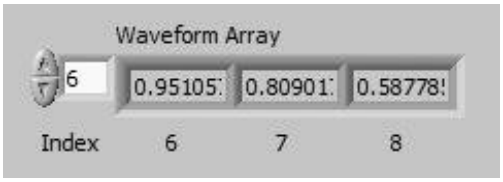
> Each iteration of the For Loop generates one point in a waveform and stores it in the waveform array created at the loop border. After the loop finishes execution, the Bundle function bundles the initial value of X, the delta value for X, and the array for plotting on the graph.

5. Return to the front panel and run the VI. Right now the graph should show only one plot.

6. Now change the delta X value to 0.5 and the initial X value to 20 and 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 (shown on the X axis).

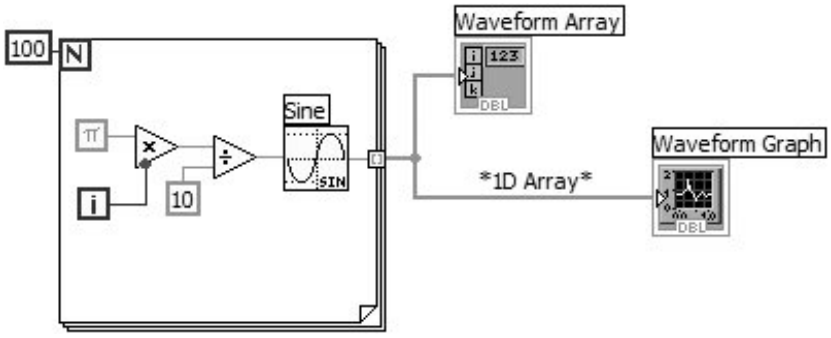
7. Place the Positioning tool on the lower-right corner of the array until the tool becomes a grid, and drag. The indicator now displays several elements with indices ascending as you go from left to right (or top to bottom), beginning with the element that corresponds to the specified index, as illustrated in Figure 8.21. (The list of indices below the indicator is a free label used for illustration purposes and won't appear on your panel.) Don't forget that you can view any element in the array simply by entering the index of that element in the index display. If you enter a number greater than the array size, the display will dim.

Figure 8.21. Waveform Array on your VI front panel after running it



In the block diagram of Figure 8.20, you specified an initial X and a delta X value for the waveform. Frequently, the initial X value will be zero and the delta X value will be 1. In these instances, you can wire the waveform array directly to the waveform graph terminal, as shown in Figure 8.22, and take advantage of the default delta X and initial X values for a graph.

Figure 8.22. Block diagram of your VI after completing step 8



8. Return to the block diagram window. Delete the Bundle function and the Numeric Constants wired to it; then select Remove Broken Wires from the Edit menu or use the keyboard shortcut (<control-B> in Windows, <command-B> in Mac OS X, and <meta-B> in Linux). Finish wiring the block diagram as shown in Figure 8.22.

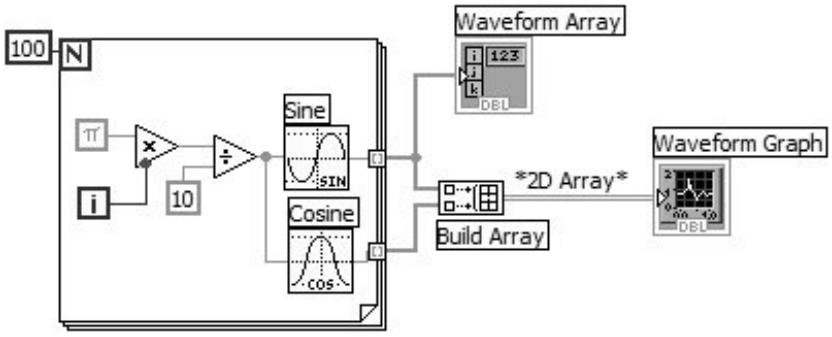
9. Run the VI. Notice that the VI plots the waveform with an initial X value of 0 and a delta X value of 1, just as it did before, with a lot less effort on your part to build the VI.

Multiple-Plot Graphs

You can also create multiple-plot waveform graphs by wiring the data types normally passed to a single-plot graph, first to a Build Array function, and then to the graph. Although you can't build an array of arrays, you can create a 2D array with each input array as a row.

10. Create the block diagram shown in Figure 8.23.

Figure 8.23. Your VI after adding an additional plot



Build Array Function

Build Array function (Programming>>Array palette) creates the proper data structure to plot two arrays on a waveform graph. Enlarge the Build Array function to include two inputs by dragging a corner with the Positioning tool. You will want to make sure you *don't* select "concatenate inputs" (just use the default value selected in the pop-up menu) to the Build Array so the output will be a 2D array. If you wish to do so, you can review the "concatenate inputs" feature of the Build Array node in Chapter 7.

Cosine Function

Cosine function (Mathematics>>Elementary & Special Functions>>Trigonometric Functions palette) computes cos (x) and returns one point of a cosine wave. The VI requires a scalar index input that it expects in radians. In this exercise, the x input changes with each loop iteration and the plotted result is a cosine wave.

11. Switch to the front panel. Run the VI. Notice that the two waveforms both 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.

12. Just to see what happens, pop up on the graph and choose Transpose Array. The plot changes drastically when you swap the rows with the columns, doesn't it? Choose Transpose Array again to return things to normal.

13. Save and close the VI. Name it Graph Sine Array.vi in your MYWORK directory or VI library.

PREV NEXT

XY Graphs

The waveform graphs you have been using are designed for plotting evenly sampled waveforms. However, if you sample at irregular intervals or are plotting a math function that has multiple Y values for every X value, you will need to specify points using their (X, Y) coordinates. *XY graphs* plot this different type of data; they require input of a different data type than waveform graphs. A single-plot XY graph and its corresponding block diagram are shown in Figures 8.24 and 8.25.

Figure 8.24. Front panel with an XY Graph displaying XY data

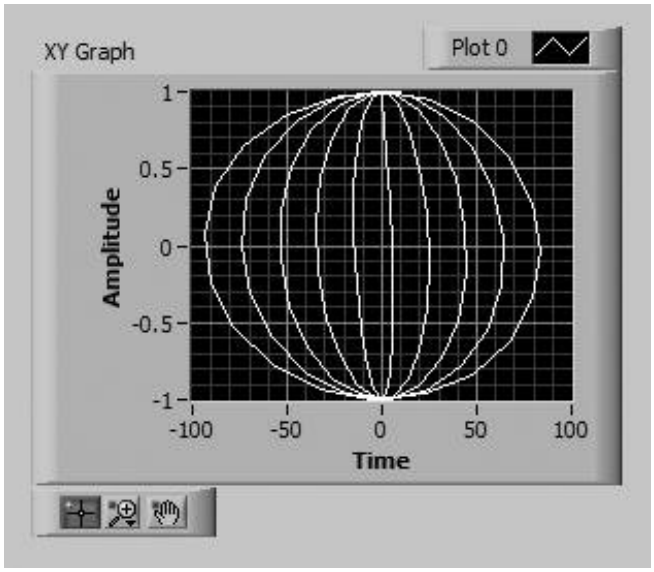
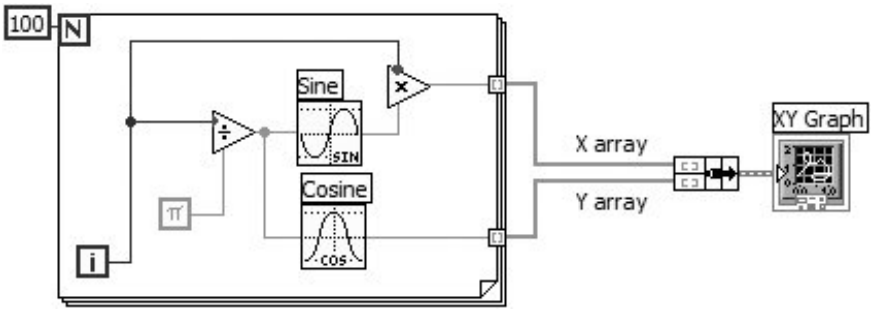


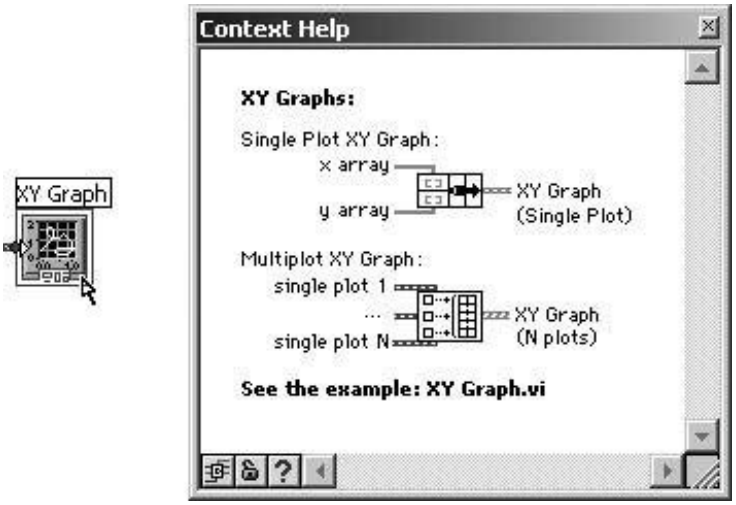
Figure 8.25. Block diagram generating XY data and displaying it in an XY Graph



For a single-plot, the XY graph expects an input of a bundled X array (the top input) and Y array (the bottom input). The Bundle function (Programming>>Cluster & Variant palette) combines the X and Y arrays into a cluster wired to the XY graph. The XY graph terminal now appears as a cluster indicator, as shown in Figure 8.25.

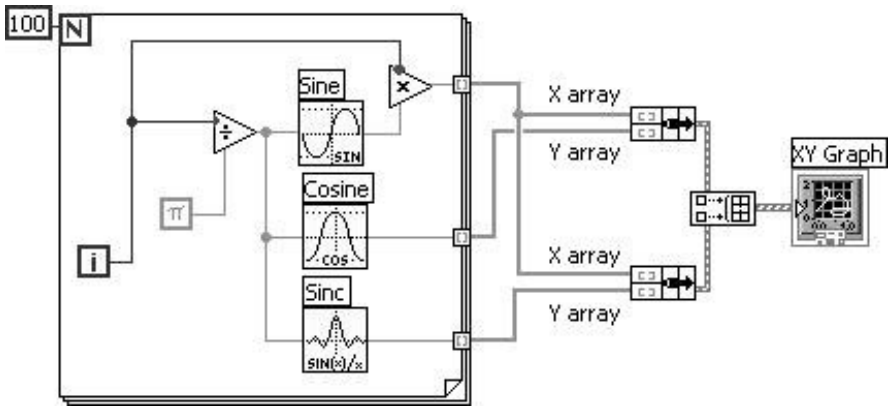
Hover over an XY graph's terminal with the Context Help window open (as shown in Figure 8.26) to see a detailed description of the single-plot and multi-plot data types that you can wire into an XY graph.

Figure 8.26. XY graph terminal's context help



For a multiple-plot XY graph, simply build an array of the single-plot clusters (of X and Y array values), as shown in Figure 8.27.

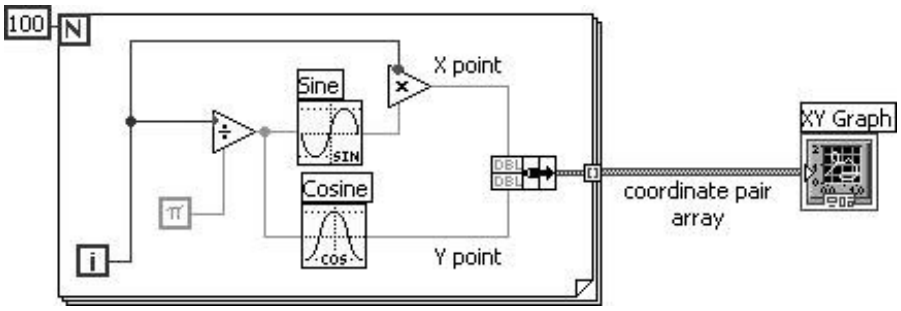
Figure 8.27. Multi-plot (1D array of single-plot clusters)



It is very easy to confuse the Bundle and Build Array functions when assembling data for graphs! Pay close attention to which one you need to use!

It is not mentioned in the Context Help window documentation for the XY graph, but you can also create a single plot from an array of XY cluster "coordinate pairs," as shown in Figure 8.28. The only downside of this single plot format is that you cannot create a multi-plot by building an array of this type of single plots.

Figure 8.28. XY coordinate pair array single plot



For some online graph examples, look at Waveform Graph.vi and XY Graph.vi in the `EVERYONE\CH08` *directory*.

Showing Optional Planes in an XY Graph

The XY graph allows you to show special grid lines, called "planes," by selecting them from its Optional Plane pop-up submenu. You can choose from *Nyquist*, *Nichols*, *S*, and *Z* planes. These planes are very useful in Radio Frequency (RF) as well as Sound and Vibration analysis, where signals are analyzed in the frequency domain, rather than time domain. You can show or hide the Cartesian (*XY*) grid lines by checking or unchecking (respectively) the Optional Plane>>Show Cartesian Lines from the XY graph's pop-up menu. The Optional Plane submenu also contains options to show and hide the plane labels and the plane lines for the selected plane. Figures 8.29 and 8.30 show XY graphs with Nyquist and Z planes visible (respectively). Note that the lines shown on the plots are grid lines, not data being displayed.

Figure 8.29. XY graph with Nyquist plane visible

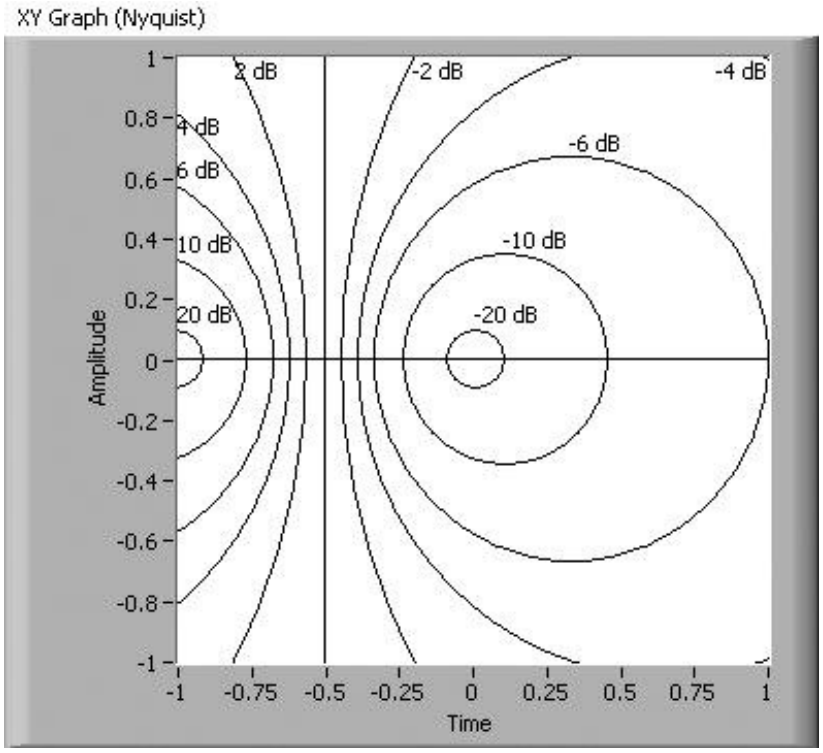
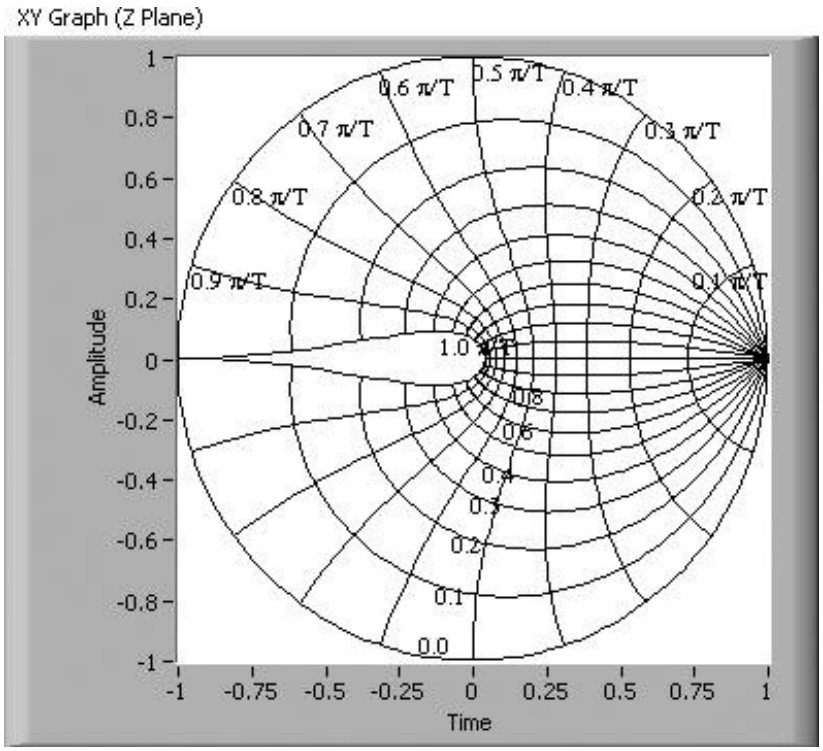


Figure 8.30. XY graph with Z plane visible



PREV NEXT

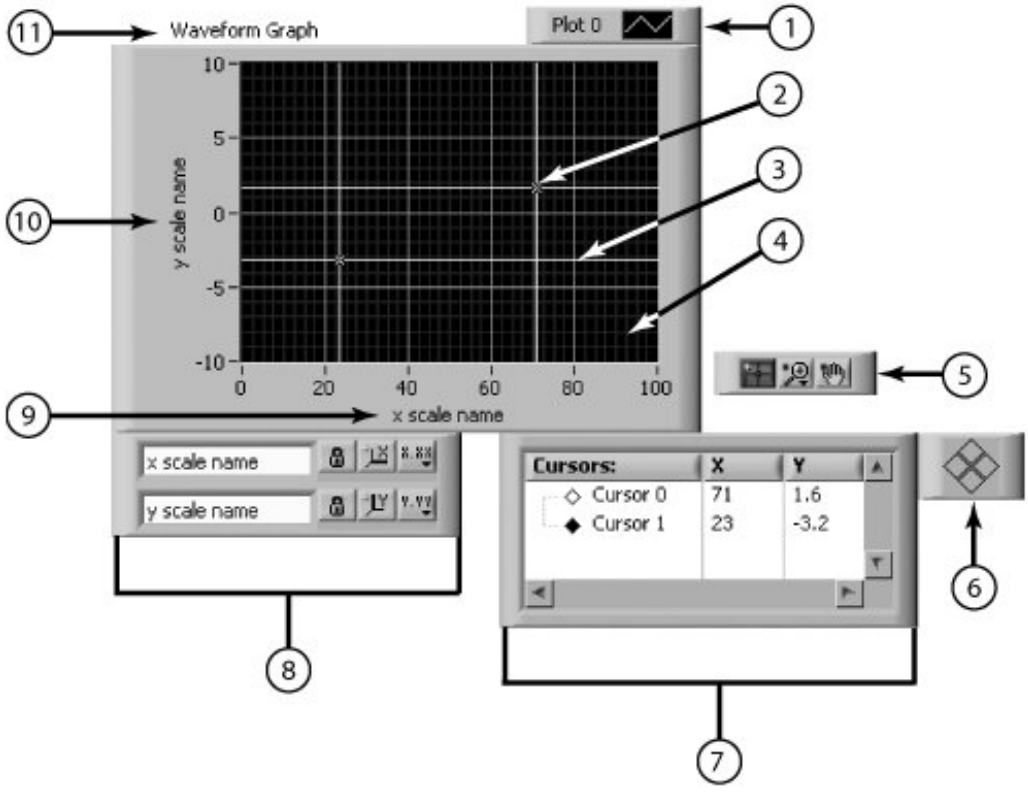
Chart and Graph Components

Graphs and charts have many powerful features that you can use to customize your plots. This section covers how to configure these options.

1 Plot legend	4 Minor-grid mark	7 Cursor legend	10 Y-scale
2 Cursor	5 Graph palette	8 Scale legend	11 Label
3 Grid mark	6 Cursor mover	9 X-scale	

Figure 8.31. Graph elements

[View full size image]



Playing with the Scales

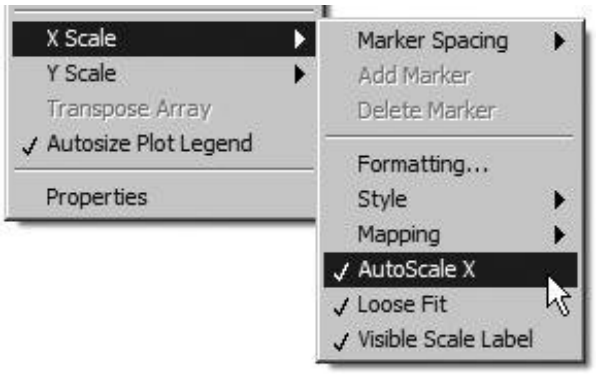
Charts and graphs can automatically adjust their horizontal and vertical scales to reflect the points plotted on them; that is, the scales adjust themselves to show all points on the graph at the greatest resolution. You can turn this *autoscaling* feature on or off using the AutoScale X and AutoScale Y options from the X Scale menu, or the Y Scale menu of the object's pop-up menu. You can also control these autoscaling features from the scale legend (which we'll get to in a minute). LabVIEW defaults to autoscaling on for graphs and off for charts. However, using autoscaling may cause the chart or graph to update more slowly, depending upon the computer and video system you use, because new scales must be calculated with each plot.

If you don't want to autoscale, you can change the horizontal or vertical scale directly using the Operating or Labeling tool to type in a new number, just as you can with any other LabVIEW control or indicator, and turn autoscaling off.

X and Y Scale Menus

The X and Y scales each have a submenu of options, as shown in Figure 8.32.

Figure 8.32. X Scale and Y Scale graph pop-up submenu options

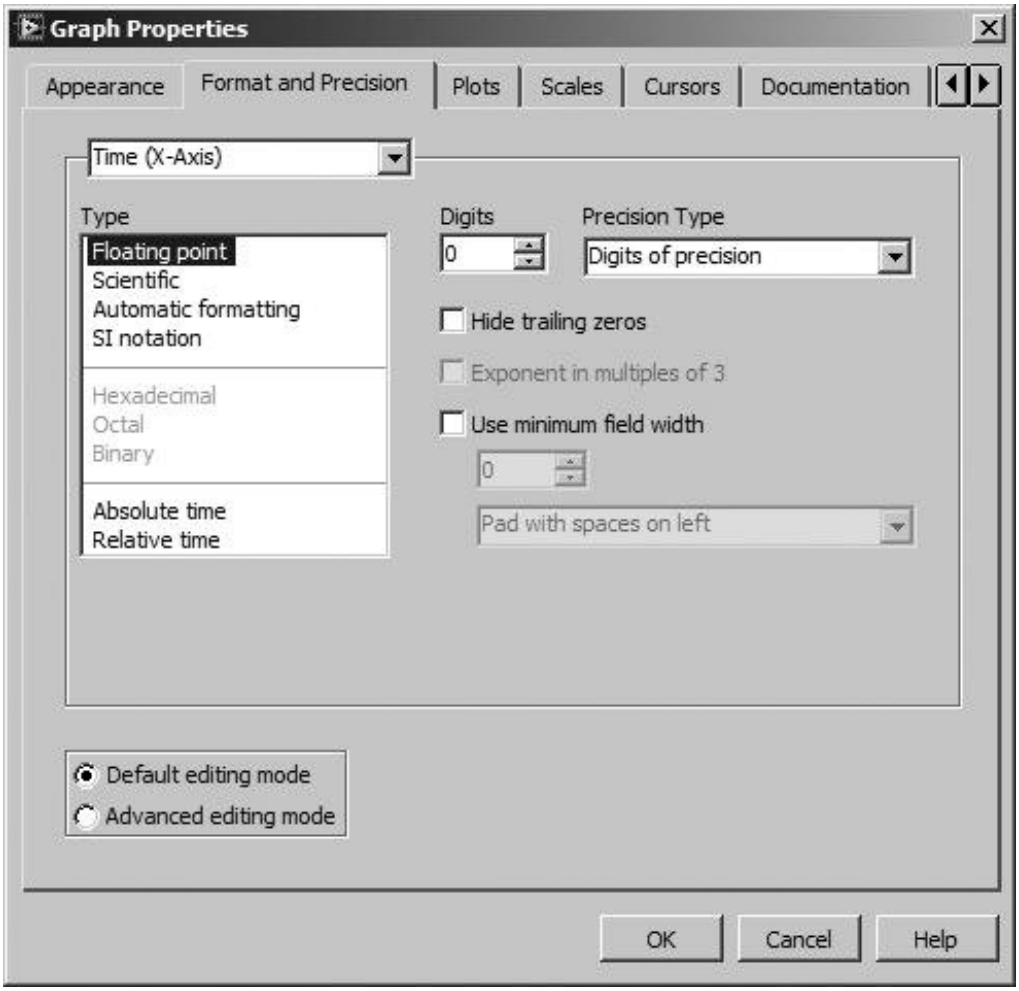


Use AutoScale to turn the autoscaling option on or off.

Normally, the scales are set to the exact range of the data when you perform an autoscale operation. You can use the Loose Fit option if you want LabVIEW to round the scale to "nicer" numbers. With a loose fit, the numbers are rounded to a multiple of the increment used for the scale. For example, if the markers increment by five, then the minimum and maximum values are set to a multiple of five instead of the exact range of the data.

The Formatting . . . option brings up a Graph Properties dialog box with the Format and Precision tab active, as shown in Figure 8.33. From here, you can configure the numeric formatting of the scale.

Figure 8.33. Format and Precision tab of the Graph Properties dialog



From the Scales tab of the Graph Properties dialog box, as shown in Figure 8.34, you can configure the following items:

- Name is the axis label.
- The Show scale label checkbox allows you to show (checked) or hide (unchecked) the axis label.
- The Log checkbox lets you to select either a linear (unchecked) or logarithmic (checked) scale for the data display.
- The Inverted checkbox allows you to define whether the axis minimum and maximum values on the scale are reversed (checked) or not (unchecked).
- Use AutoScale to turn the autoscaling option on or off. If Autoscale is off, then you can use

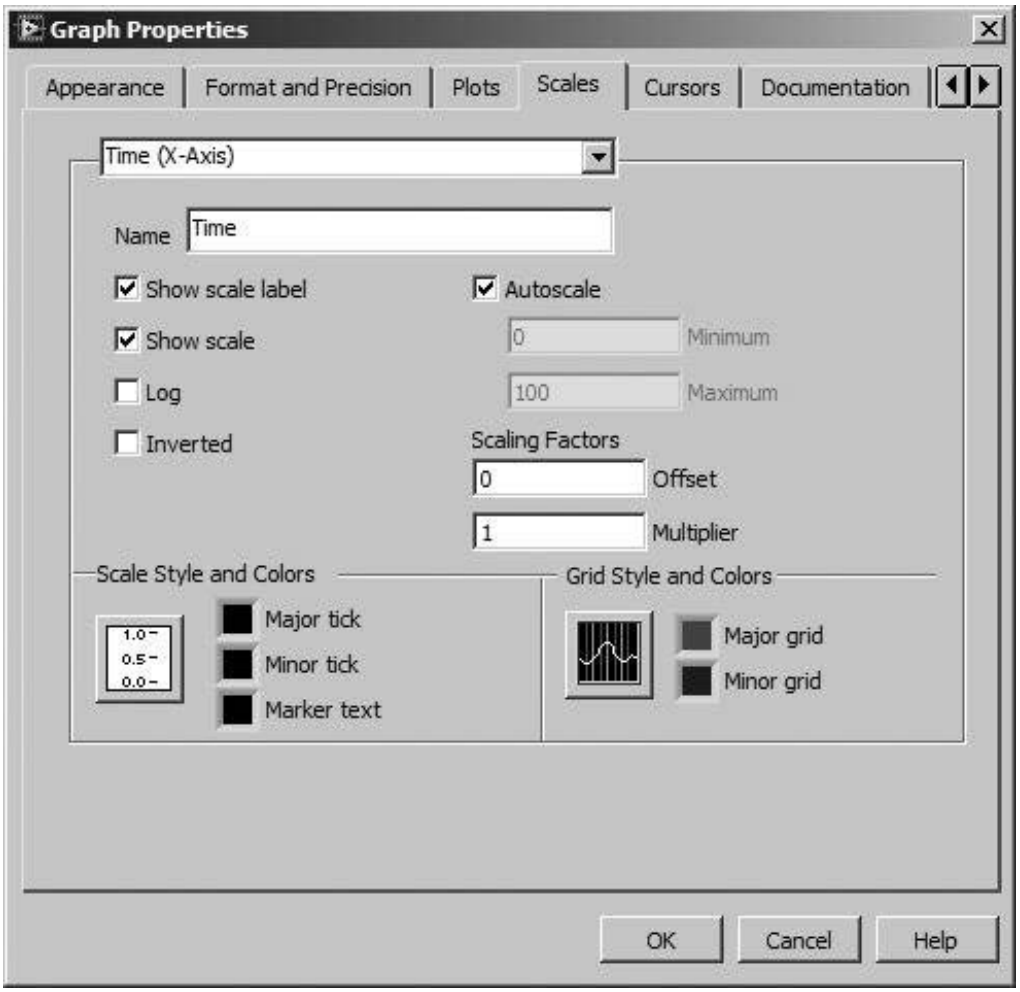
Minimum and Maximum to set the scale range.

- The Scale Style and Colors menu lets you select major and minor tick marks for the scale, or none at all. Click on this icon to see your choices. A major tick mark corresponds to a scale label, while a minor tick mark denotes an interior point between labels. This menu also lets you select the markers for a given axis as either visible or invisible.

- The Grid Style and Colors lets you choose between no grid lines only at major tick mark locations, or grid lines at both major and minor tick marks. You can also change the color of the grid lines here. Click on the grid buttons to see a pullout menu of your options.

- You can set Offset, the *X* value you want to start at, and Multiplier, the increment between *X* values (same as delta *X*), in the Scaling Factors section.

Figure 8.34. Scales tab of the Graph Properties dialog



The Scale Legend

The *scale legend* lets you create labels for the *X* and *Y* scale (or for multiple *XY* scales, if you have more than one), and have easy pop-up access to their configuration. The scale legend gives you buttons for scaling the *X* or *Y* axis, changing the display format, and autoscaling.

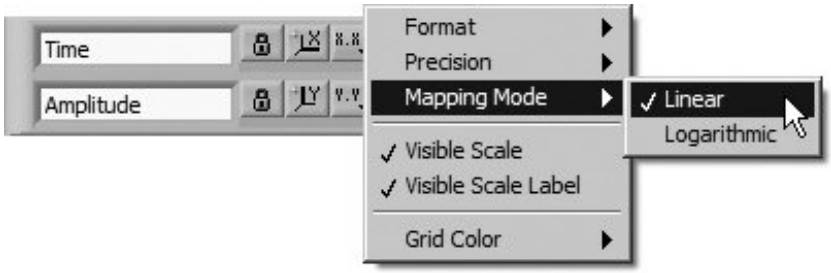
Popping up on the graph or chart and choosing Visible Items>>Scale Legend will give you a box like the one shown in Figure 8.35.

Figure 8.35. Scale legend

In the scale legend, you can type the names of your scales in the textbox. This text will show up on the graph or chart's box *X* or *Y* axis.

You can click on the buttons to configure the same options as in the X|Y Scale Formatting pop-up window just described (see Figure 8.36). It's just a more convenient way to access this information for some people.

Figure 8.36. Scale legend buttons pop-up menu options



Scale Lock Button

Use the Operating tool to click on the Scale Lock button to toggle autoscaling for each scale, the visibility of scales, and so on.

You can resize the scale legend with the Positioning tool, just like an array, to create, show, or hide additional scales on your chart or graph.

The Plot Legend

Charts and graphs use a default style for each plot unless you have created a custom plot style for it. The *plot legend* lets you label, color, select line style, and choose point style for each plot. Show or hide the legend using the Visible Items>>Plot Legend option of the chart or graph pop-up menu. You can also specify a name for each plot in the legend. An example of a legend is shown in Figure 8.37.

Figure 8.37. Plot Legend

When you select Plot Legend, only one plot shows in the box that appears. You can show more plots by dragging down a corner of the legend with the Positioning tool. After you set plot characteristics in the Plot Legend, the plot retains those settings, regardless of whether the legend is visible. If the chart or graph receives more plots than are defined in the legend, LabVIEW draws the extra plots in default style.

When you move the chart or graph body, the legend moves with it. You can change the position of the legend relative to the graph by dragging only the legend to a new location. *Resize the legend on the left to give labels more room in the window or on the right to give plot samples more room*.

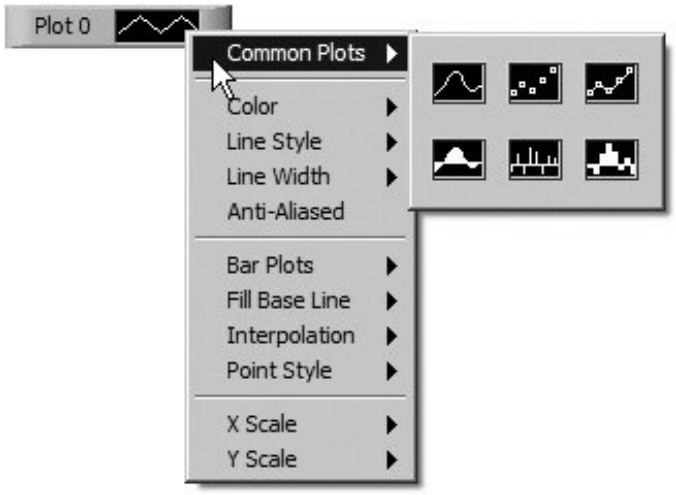
By default, the plot legend of a graph or chart will automatically resize to the width of the longest plot name visible in the legend. This behavior can be disabled by popping up on the graph or chart and selecting Autosize Legend from the shortcut menu to remove the

checkmark.

By default, each plot is labeled with a number, beginning with zero. You can modify this label the same way you modify other LabVIEW labelsjust start typing with the Labeling tool. Each plot sample has its own menu to change the plot, line, color, and point styles of the plot. You can also access this menu by clicking on the legend with the Operating tool.

- The Common Plots option lets you easily configure a plot to use any of six popular plot styles, including a scatter plot, a bar plot, and a fill to zero plot (see Figure 8.38). Options in this subpalette configure the point, line, and fill styles of the graph in one step (instead of setting these options individually, as listed next). Look at Activity 8-6 for an example of a bar graph.

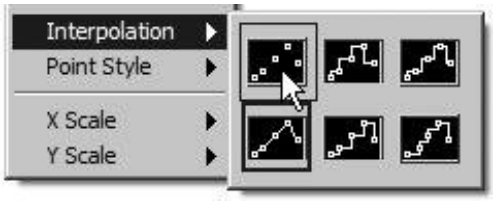
Figure 8.38. Set the appearance of a plot by choosing from one of the Common Plots in the plot legend



- The Color option displays the Color palette so that you can select the plot color. You can also color the plots on the legend with the Color tool. You can change the plot colors while running the VI.
- The Line Style and Line Width offer different types of patterns and widths for the lines to plot.
- The Anti-Aliased option makes line plots appear smoother and nicer overall, but be aware that anti-aliased line drawing can be computationally intensive and slow performance.
- The Bar Plots option lets you create bar plots of 100%, 75%, or 1% width, either horizontally or vertically. The Fill Baseline option controls the baseline of the bars; plots can have no fill or they can fill to zero, negative infinity, or infinity. (With multi-plots, it is possible to fill to the value of another plot.)
- The Interpolation option determines how LabVIEW draws lines between data points (see Figure 8.39). The first option does not draw any lines, making it suitable for a scatter plot (in

other words, you get points only). The option at the bottom left draws a straight line between points. The four stepped options link points with a right-angled elbow, which is useful for histograms.

Figure 8.39. Choosing an Interpolation option to define how lines will connect a plot's data points



- The Point Style options display different point styles (none, round, square, hollow, filled, etc.) you can choose from.

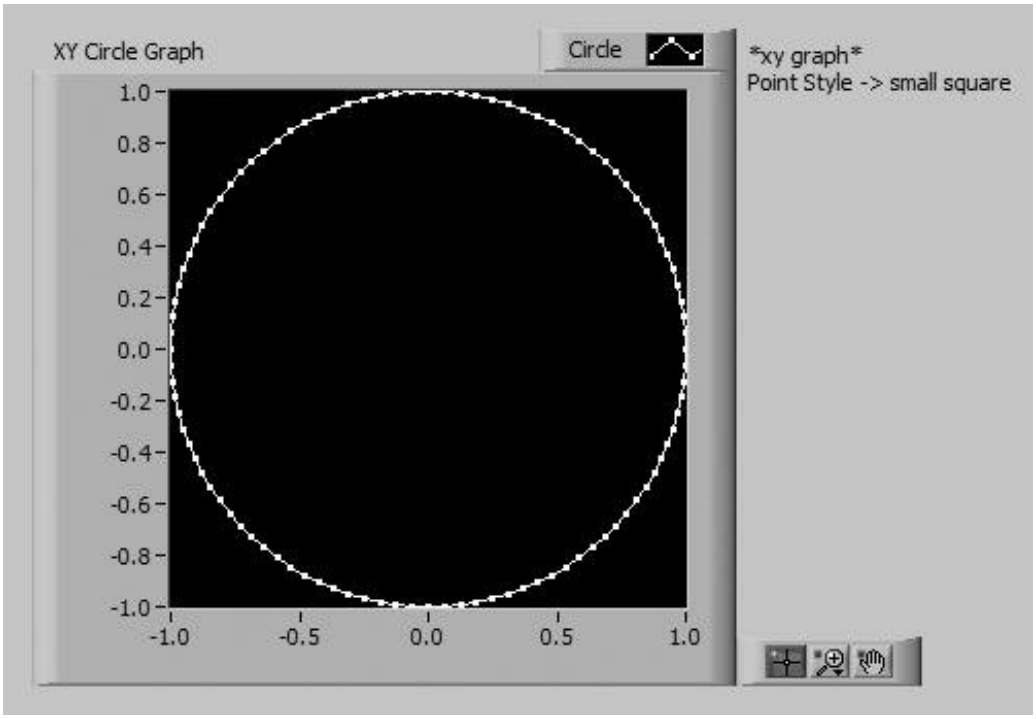
PREV NEXT

Activity 8-3: Using an XY Graph to Plot a Circle

You will build a VI that plots a circle.

1. Open a new front panel. You will recreate the panel shown in Figure 8.40.

Figure 8.40. Front panel of the VI you will create during this activity



2. Place an XY Graph (Graph palette) onto your VI's front panel. Label the graph `XY Circle Graph`.

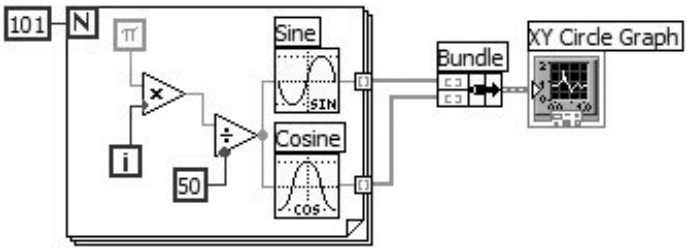
3. Enlarge the graph by dragging a corner with the Positioning tool. Try to make the plot region approximately square.

You can maintain the aspect ratio while dragging by holding down the <shift> key and dragging diagonally.

4. Pop up on the graph and select Visible Items>>Plot Legend. Resize the legend from the left side and enter the `Circle` label using the Labeling tool. Pop up on the line in the legend and select the small square from the Point Style palette. Then select a new plot color from the Color palette.

5. Build the block diagram shown in Figure 8.41.

Figure 8.41. Block diagram of the VI you will create during this activity



Sine Function

Cosine Function

Sine and Cosine functions (Mathematics>>Elementary & Special Functions>> Trigonometric Functions palette) calculate the sine and cosine, respectively, of the input number. 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 and one cycle of a cosine wave.

Bundle Function

Bundle function (Programming>>Cluster & Variant palette) assembles the sine array (X values) and the cosine array (Y values) to plot the sine array against the cosine array.

Pi Constant

Pi constant (Programming>>Numeric>>Math & Scientific Constants palette) is used to provide radian input to Sine and Cosine functions.

6. Return to the front panel and run the VI. Save it as Graph Circle.vi in your `MYWORK` directory or VI library. Congratulations! You're getting the hang of this stuff!

Using the Graph Palette

Standard Operate Mode

Zoom Button

The *graph palette* is the little box that contains tools allowing you to pan (i.e., scroll the display area), focus in on a specific area using palette buttons (called zooming), and move cursors around (we'll talk about cursors shortly). The palette, which you can make visible by selecting Graph Palette from the Visible Items submenu of the chart or graph pop-up menu, is shown in Figure 8.42.

Figure 8.42. Graph palette

Pan Button

The remaining three buttons let you control the operation mode for the graph. Normally, you are in standard mode, meaning that you can click on the graph cursors to move them around. If you press the pan button, you switch to a mode where you can scroll the visible data by dragging sections of the graph with the pan cursor. If you click on the zoom button, you get a pop-up menu that lets you choose from several methods of zooming (focusing on specific sections of the graph by magnifying a particular area).

Here's how these zoom options work:

Zoom by Rectangle

Zoom by Rectangle. Drag the cursor to draw a rectangle around the area you want to zoom in on. When you release the mouse button, your axes will rescale to show only the selected area.

Zoom by Rectangle in X

Zoom by Rectangle in X, with zooming restricted to X data (the Y scale remains unchanged).

Zoom by Rectangle in Y

Zoom by Rectangle in Y, with zooming restricted to Y data (the X scale remains unchanged).

Zoom to Fit

Zoom to Fit. Autoscales all X and Y scales on the graph or chart, such that the graph shows all available data.

Zoom In About a Point

Zoom In About a Point. If you hold down the mouse on a specific point, the graph will continuously zoom in until you release the mouse button.

Zoom Out About a Point

Zoom Out About a Point. If you hold down the mouse on a specific point, the graph will continuously zoom out until you release the mouse button.

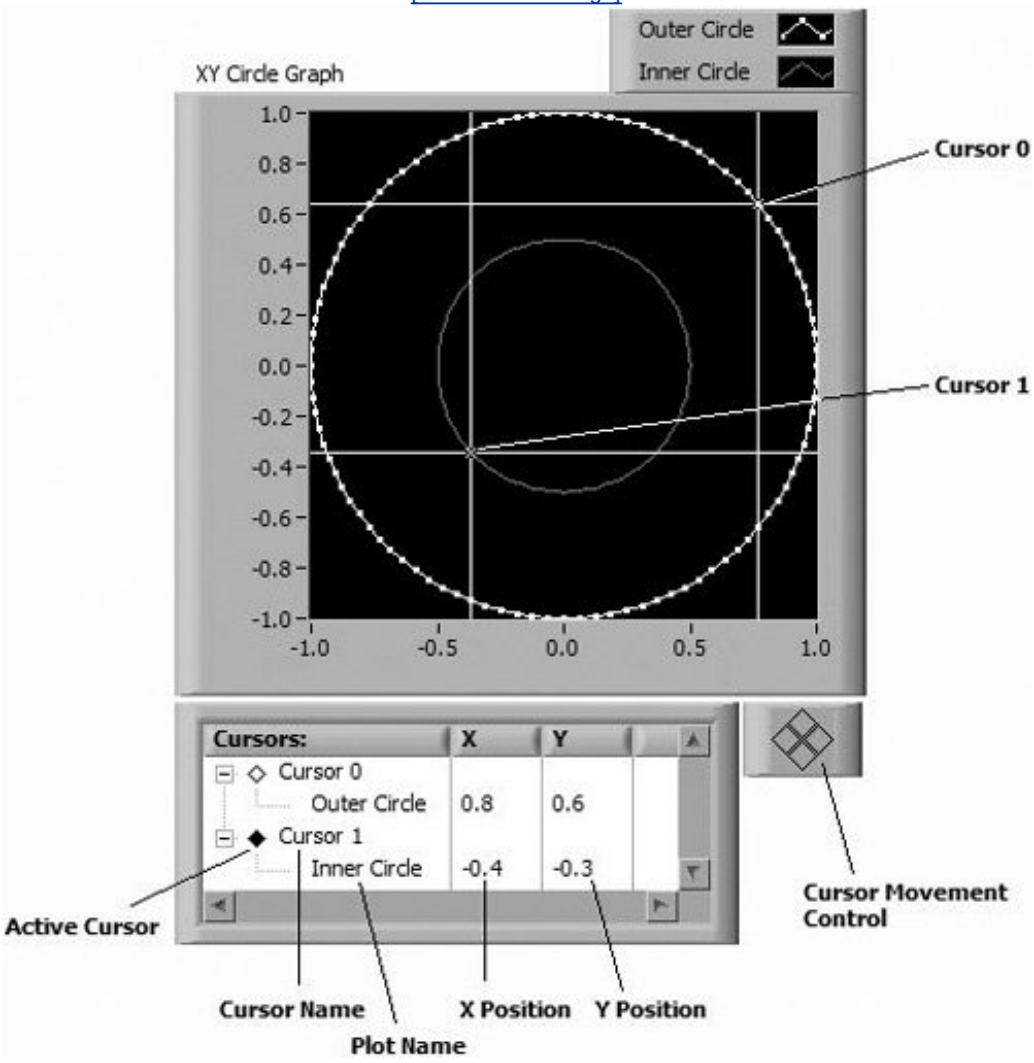
For the last two modes, Zoom In and Zoom Out About a Point, shift-clicking will zoom in the other direction. Note that this feature requires a VI to be in run mode, and will not work when a VI is in edit mode.

Graph Cursors

LabVIEW graphs have *cursors* to mark data points on your plot to further animate your work. Figure 8.43 shows a picture of a graph with cursors and the cursor legend visible. You won't find cursors on charts, so don't bother to look.

Figure 8.43. Graph cursors

[View full size image]



You can view the Cursor palette by selecting Visible Items>>Cursor Legend from the graph's pop-up menu. When the cursor legend first appears, it is empty.

Pop up on the cursor legend and select a mode from the Create Cursor>> shortcut menu. The following cursor modes are available:

- Free mode allows the cursor to be moved freely within the plot areathe cursor is not locked to

points of a plot.

- Single-Plot mode allows the cursor to be moved only to points on the plot that is associated with the cursor. To associate the cursor with a plot, right-click the cursor legend row and select the desired plot from the Snap To shortcut menu.
- Multi-Plot mode is only valid for mixed signal graphs (which we will discuss in the "Mixed Signal Graphs" section). It allows the cursor to be moved only to points in the plot area. The cursor will report values at the specified x-value for all the plots with which the cursor is associated. To associate the cursor with a plot, right-click the cursor legend row and select the desired plot from the Snap To shortcut menu.

You cannot change a cursor's mode after it is created. If you want a cursor of a different mode, you must first delete the existing cursor and then create another cursor of the desired mode.

Active Cursor (Inactive)

A graph can have as many cursors as you want. The cursor legend helps you keep track of them, and you can stretch it to display multiple cursors.

What does this cursor legend do, you ask? You can label the cursor in the first column of the cursor legend. The next column shows the *X* position, and the next after that shows the *Y* position.

Active Cursor (Active)

Click on the Active Cursor diamond to the left of the cursor name to make a cursor active, and selected for movement (the diamond will be solid black when the cursor is active and unfilled (white) when the cursor is inactive). Note that more than one cursor can be active at the same time.

Cursor Movement Control

When you click a button on the Cursor Movement Control, all active cursors move. You can move the active cursors up, down, left, and right. You can also move cursors around programmatically using property nodes, which you'll learn about in Chapter 13.

You can also drag a cursor around the graph with the Operating tool. If you drag the intersection point, you can move the cursor in all directions. If you drag the horizontal or vertical lines of the

cursor, you can drag only vertically or horizontally, respectively.

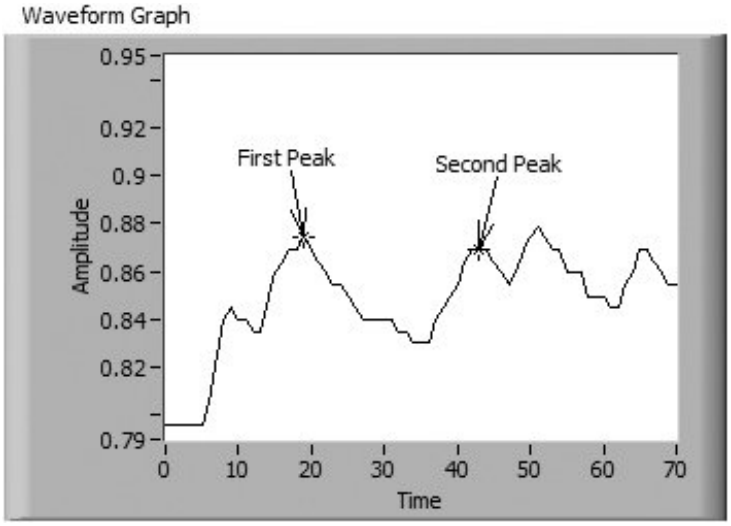
Pop up on the cursor label inside the cursor legend to attributes like cursor style, point style, and color. You can also select a plot from the Snap To>> pop-up submenu to lock the cursor to a plot. Locking restricts cursor movement so it can reside only on plot points, but nowhere else in the graph.

To delete a cursor, pop-up on the cursor label in the cursor legend and choose Delete Cursor.

Graph Annotations

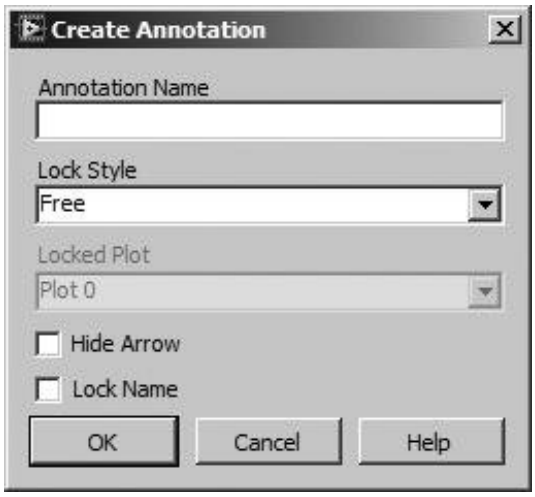
Graph annotations are useful for highlighting data points of interest on a graph. An annotation appears as a labeled arrow that can be used to describe features in your data, as shown in Figure 8.44.

Figure 8.44. Graph annotations



You can create and modify annotations interactively with the mouse, as well as programmatically using property nodes, which are discussed in Chapter 13. When the VI is in *run mode*, you can create an annotation by selecting Create Annotation from the graph's pop-up menu (in *edit mode*, all annotation options are available from the graph's Data Operations>> pop-up submenu). Selecting Create Annotation will open the Create Annotation dialog, which is used for defining the new annotation's label (Annotation Name) and some basic attributes, as shown in Figure 8.45.

Figure 8.45. Create Annotation dialog



Annotation Cursor

Annotations have three parts: the label, the arrow, and the cursor. The label and arrow are easy to identify, but the annotation cursor is a little bit harder to seeit's right under the tip of the annotation's arrow. Popping up on an annotation's cursor provides access to options for editing the annotation's properties, as well as deleting the annotation.

An annotation's label may be moved relative to its cursor by clicking on it and dragging it to a new location. When moved, the arrow will always point from the annotation's label to its cursor. You can also move the annotation cursor, depending on the setting of its Lock Style attribute. The Lock Style can be one of the following:

- Free allows the annotation cursor to be moved to anywhere on the graph.
- Snap to All Plots allows the annotation cursor to be moved to any data point of any plot.
- Snap to One Plot allows the annotation cursor to be moved to any data point on the plot specified by Locked Plot.

Finally, to delete an annotation, select the Delete Annotation option from the annotation cursor's pop-up menu. You can also delete all annotations on a graph by selecting the Delete All Annotations option from the graph's pop-up menu.

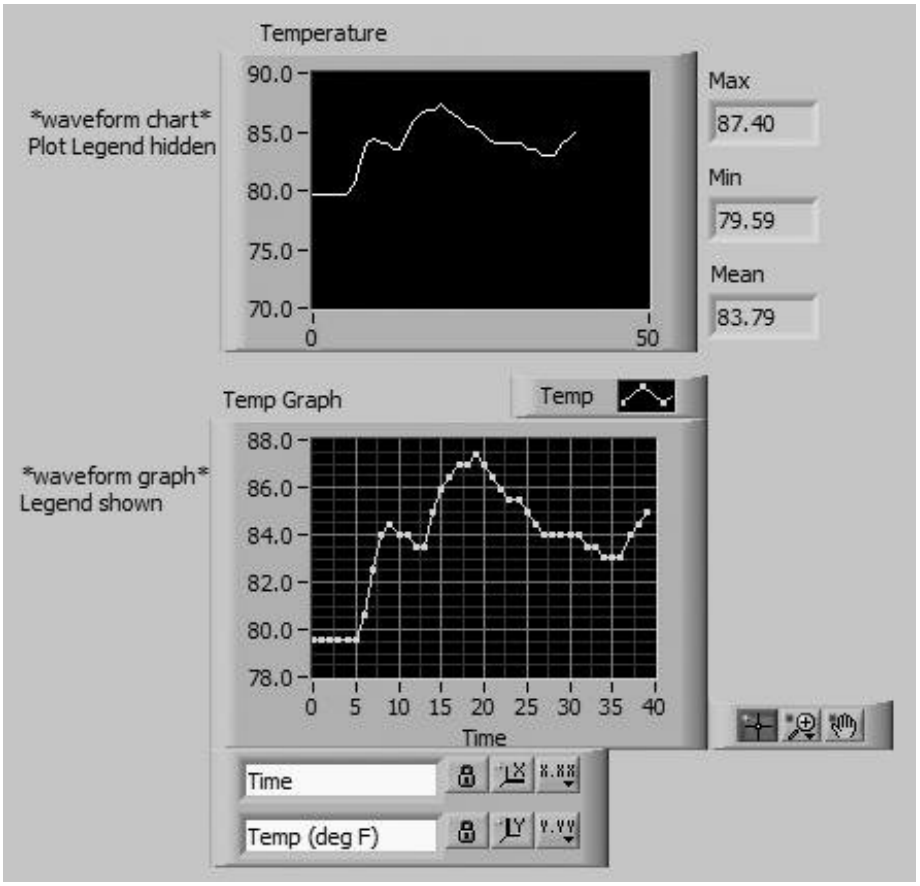
PREV NEXT

Activity 8-4: Temperature Analysis

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

1. Open a new front panel and build the VI shown in Figure 8.46.

Figure 8.46. Front panel of the VI you will create during this activity



2. Rescale the chart so that it ranges from 70.0 to 90.0. Also, make sure autoscaling is on for both axes of the graph (it should be, because that is the default).

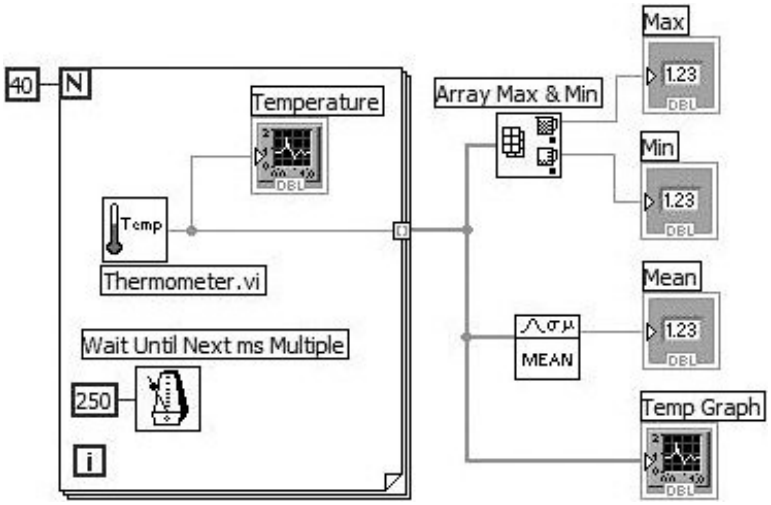
3. In the plot legend, using the Labeling tool, type in `Temp` as shown. Now pop up (or click with the Operating tool) on the `Temp` plot representation in the legend and change the Point Style to

small squares. Feel free to color your plots as well.

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

4. Build the block diagram shown in Figure 8.47. Make sure to use the Help window to show you the inputs and outputs of these functions, and pay attention to the wire whiskers and tip strips, or you will almost certainly wire to the wrong terminal!

Figure 8.47. Block diagram of the VI you will create during this activity



Thermometer.vi

Thermometer.vi. Use the Select A VI . . . palette to access the one you built. It should probably be in your `MYWORK` directory. If you don't have it, you can use Thermometer.vi in the `EVERYONE\CH05` directory or Digital Thermometer.vi, located in the `<LabVIEW>\Activity` folder. Thermometer returns one temperature measurement each time it is called.

Wait Until Next ms Multiple Function

Wait Until Next ms Multiple function (Programming>>Timing palette) causes the For Loop to execute every 0.25 seconds (250 ms).

Array Max & Min Function

Array Max & Min function (Programming>>Array palette) returns the maximum and minimum values in the array; in this case, the maximum and minimum temperatures measured during the acquisition.

Mean.vi

Mean.vi (Mathematics>>Probability and Statistics palette) returns the average of the temperature measurements. Make sure you wire to the right output terminal to get the mean.

5. The For Loop executes 40 times. The Wait Until Next ms Multiple function causes each iteration to take place approximately every 250 ms. The VI stores the temperature measurements in an array created at the For Loop border using auto-indexing. After the For Loop completes, the array passes to various nodes. The Array Max & Min function returns the maximum and minimum temperature. Mean.vi returns the average of the temperature measurements.

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

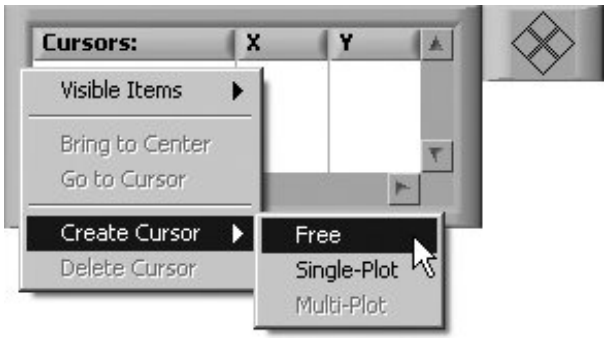
7. Using the scale legend (show it by choosing Visible Items>>Scale Legend from the pop-up menu on the graph), change the precision so that the graph shows three decimal places on the Y scale.

8. Using the Graph Palette, click on the Zoom button, select a zooming mode, and zoom in on the graph.

   
   Zoom Button

9. Pop up on the graph and select Visible Items>>Cursor Legend. The cursor legend will first appear empty (with no cursors). Pop up on the cursor legend and select Create Cursor>>Free to create a *free cursor* (one that does not snap to plot points), as shown in Figure 8.48.

Figure 8.48. Creating a free cursor from the pop-up menu of the cursor legend



Use the Operating tool to drag the cursor around on the graph; pay attention to how the *X* and *Y* values in the cursor display change. These values can help you determine the value of a particular point on the graph. Now use the Cursor Movement Control buttons to move the cursor around. Create a second cursor and make it active. Use the Cursor Movement Control to move both cursors at the same time. Change the color of one of your cursors by popping up on the cursor label in the cursor legend and selecting a color from the Attributes>>Color submenu.

Are you starting to see what cursors can do? If you want, select the `Cursor 0` text and replace it with a cursor name of your own. Next, let's create a cursor that snaps to the data points of a plot by selecting Create Cursor>>Single-Plot. Notice that this cursor can be dragged around on the graph with the mouse, but it snaps to points on a plotit cannot be moved to arbitrary locations. You can tell which plot a cursor is snapped to, by looking in the cursor legend at the plot name beneath the cursor label. You can configure a cursor to snap to All Plots or only one specific plot by popping up on the cursor label and selecting All Plots or the desired plot name from the Snap To>> submenu. When you use the Cursor Movement Control on a cursor, you'll find that the cursor will track the plot it's assigned to.

10. Close and save the VI. Name it Temperature Analysis.vi and place it in your `MYWORK` directory or VI library.

PREV NEXT

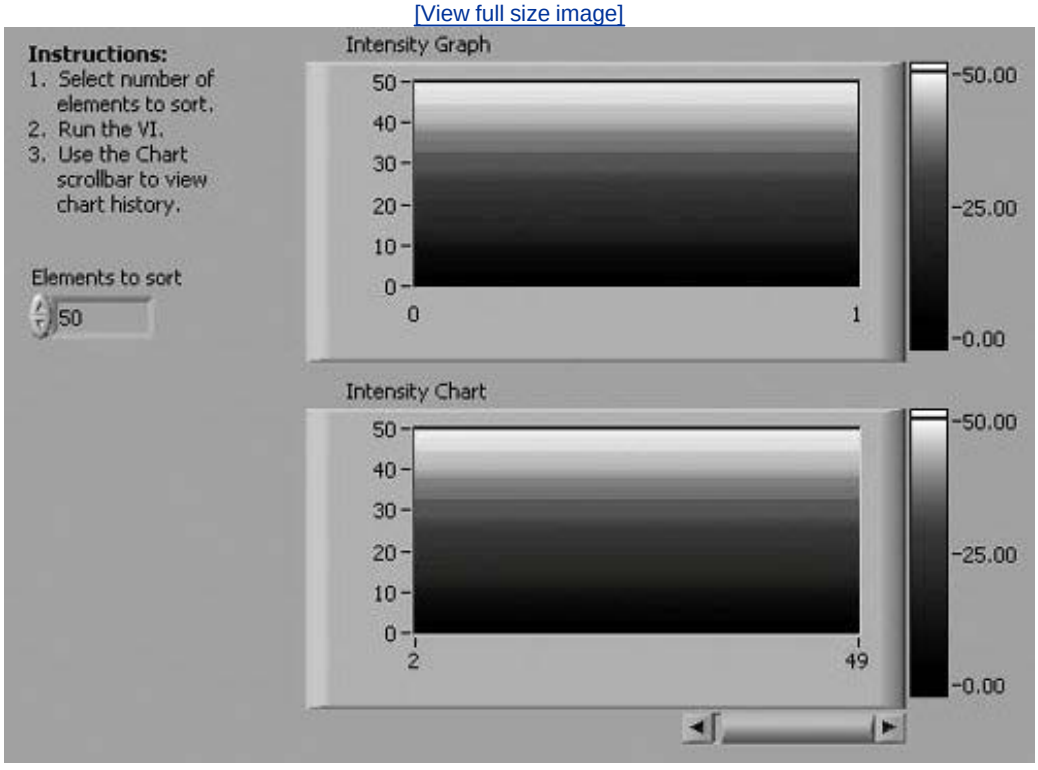
Intensity Charts and GraphsColor as a Third Dimension

So what do you do if you want to plot three variables against each other, instead of just two? If you're using Windows, you can use the ActiveX 3D Graph Controls, which we'll discuss next.

On all versions of LabVIEW, you can also make an intensity plot. *Intensity charts and graphs* display three dimensions of data on a 2D plot by using color to display values of the third dimension of data (the "Z" values). Like the waveform chart, the intensity chart features a scrolling display, while the intensity graph display is fixed. Intensity plots are extremely useful for displaying patterned data such as terrain, where color represents altitude over a two-dimensional area, or temperature patterns, where color represents temperature distribution (see Figure 8.49).

Figure 8.49. Intensity Graph (top) and Intensity Chart (bottom), which display color as a third dimension

[View full size image]

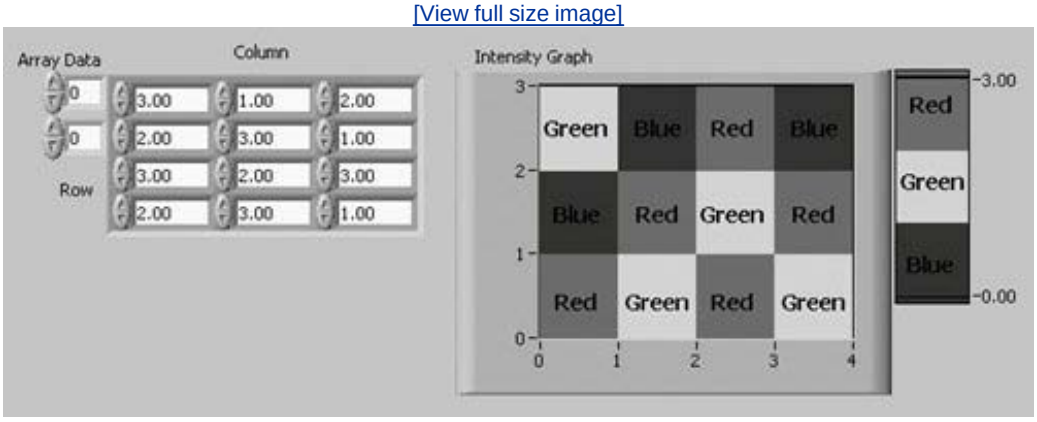


Intensity plots function much like two-dimensional charts and graphs in most ways, with the addition of color to represent the third variable. A color scale is available so you can set and display the color mapping scheme. Intensity graph cursor displays also include a *Z* value.

Intensity charts and graphs accept 2D arrays of numbers, where each number in the array is a color value. The indices of each array element represent the plot location for that color. Not only can you define the mapping of numbers to colors using the color scale (which works like a color ramp, if you've played with that), but you can also do it programmatically using property nodes (Chapter 12, "Instrument Control in LabVIEW," will tell you all about them). The simple example in Figure 8.50 shows a 3 x 4 array plotted on an intensity graph. The colors mapped are blue (1.0), green (2.0), and red (3.0).

Figure 8.50. An intensity graph displaying a 2D array of intensity values as colors

[View full size image]



To assign a color to a value on the color scale, pop up on the corresponding marker and select Marker Color>> ("markers" are the numeric displays on the *Z* scale legend). The *Z* scale of intensity charts and graphs has Arbitrary Marker Spacing by default, so you can alter the "gradient" of colors by dragging a marker with the Operating tool. Create new markers by selecting Add Marker from the Color Scale pop-up menu, and then drag them to a desired location and assign a new color. If you want to learn more about intensity charts and graphs, we suggest playing around with them. You can also look in LabVIEW's online documentation for more exact instructions.

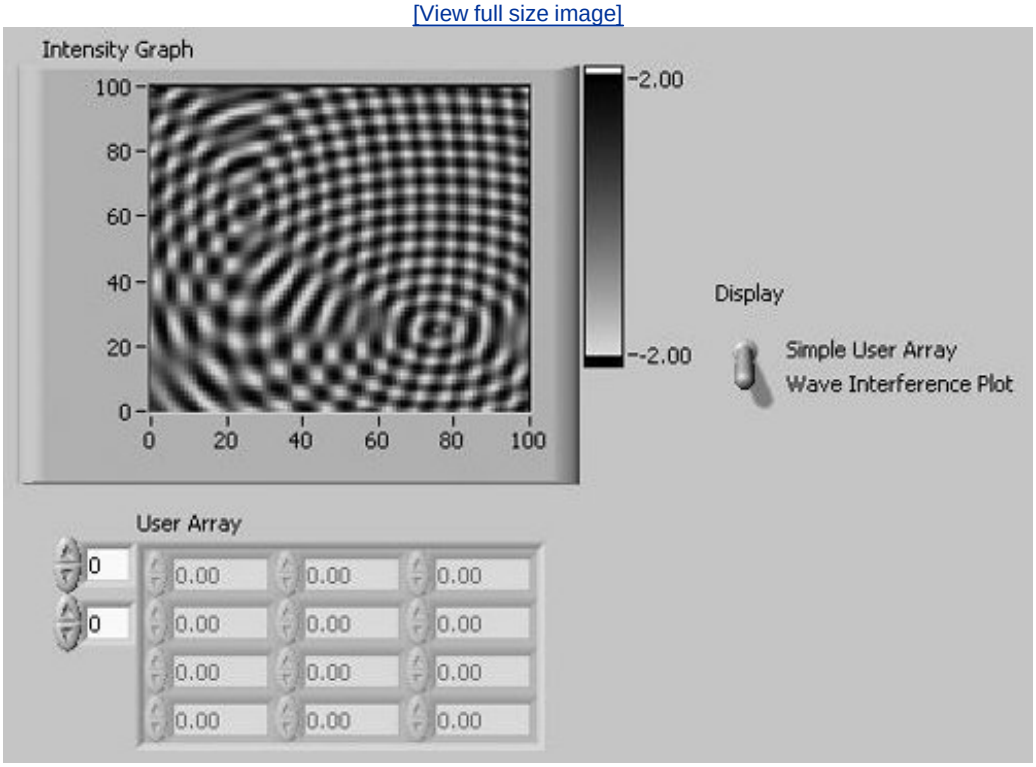
Activity 8-5: The Intensity Graph

In this activity, you will look at a VI that displays wave interference patterns. This VI will also show you how the intensity graph maps its input 2D array to the display.

1. To get an idea of how intensity graphs and charts work, open the Intensity Graph Example VI, located in the `EVERYONE\CH8` directory. Run the VI. You will see an intricate interference waveform plotted on the graph (see Figure 8.51). The color range is defined in the block diagram using the intensity graph attribute node. Modify the color range by clicking on the color boxes in the first frame of the Sequence Structure with the Operating tool, and then choosing a new color from the palette that appears. Run the VI again.

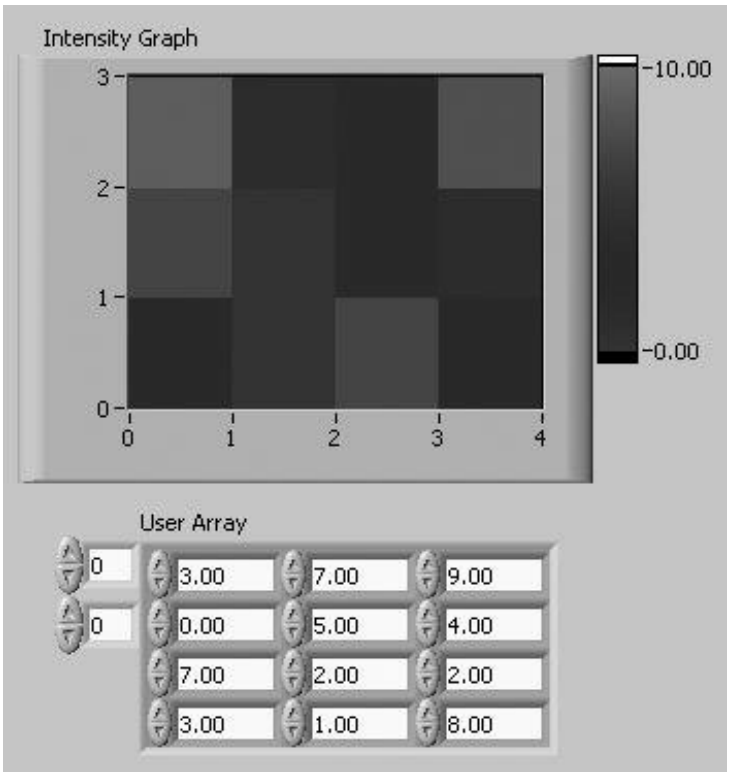
Figure 8.51. Intensity Graph Example.vi front pane after running in "Wave Interference Plot" mode

[View full size image]



2. Switch the `Display` switch to Simple User Array and enter values between 0.0 and 10.0 in the `User Array` control (the color range for the graph has been defined in the block diagram using the intensity graph property nodeblue (0.0) to red (10.0)). After you enter all the values, run the VI. Notice how the magnitude of each element is mapped to the graph. Now change your values and run it again (see Figure 8.52).

Figure 8.52. Intensity Graph Example.vi front panel after running in "Simple User Array" mode



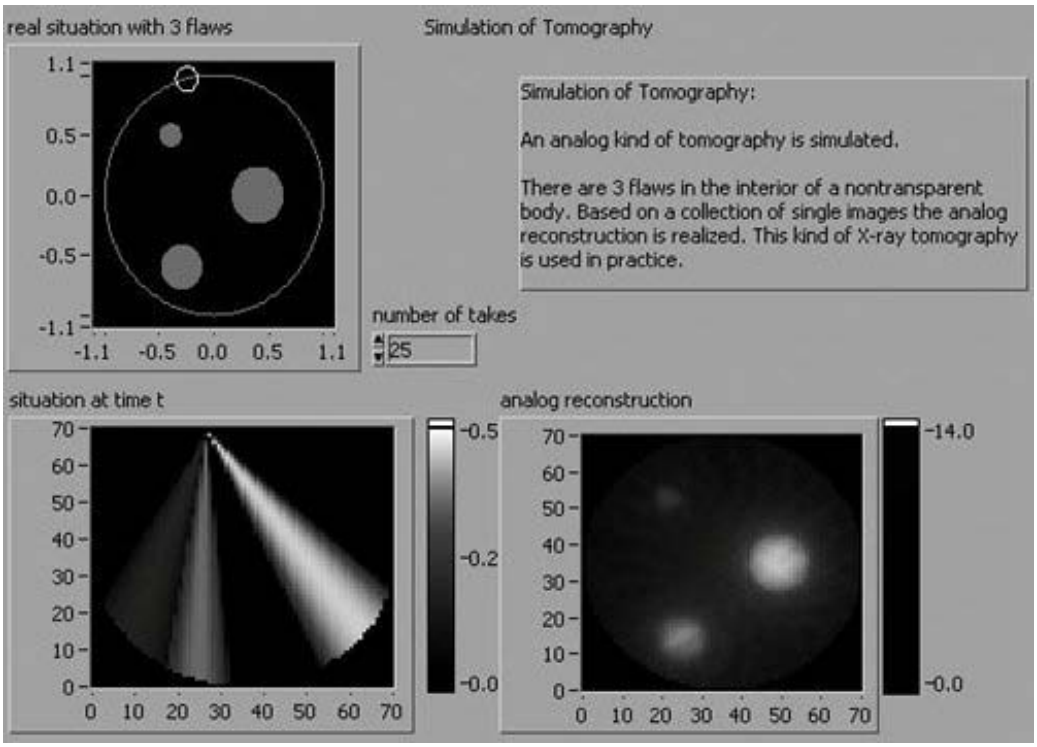
3. Take a look at the block diagram to see how the VI works.

4. Close the VI and don't save any changes.

For more examples of intensity plots, open some of the VIs that ship with LabVIEW in `examples\general\graphs\intgraph.llb`. You can also check out Simulation of Tomography and Heat Equation Example in `examples\analysis\mathxmpl.llb` (see Figure 8.53).

Figure 8.53. Simulation of Tomography.vi front panel

[View full size image]



3D Graphs

For true three-dimensional graphs, LabVIEW for Windows[1] Professional Version offers a 3D Surface Graph, 3D Parametric Graph, and 3D Curve Graph from the Graph palette.

[1] If you're a Mac or Linux user and feel left out, let National Instruments know you'd like this feature across all platforms.

These 3D graphs are only for Windows, and are not present in the Base package of LabVIEW.

Figures 8.54 and 8.55 show a VI's front panel and block diagram that presents an example of a "doughnut"called a torusplotted on the 3D Parametric Graph control. Unlike the previous charts and graphs you've seen, 3D graphs don't have a simple block diagram terminal; they actually use special subVI functions that are automatically created when you drop a 3D graph on the front panel. (You can find the Torus.vi in the LabVIEW examples, using the NI Example Finder.)

Figure 8.54. Torus.vi front panel showing a 3D toroidal "doughnut" surface plot . . . mmmm doughnuts!

[View full size image]

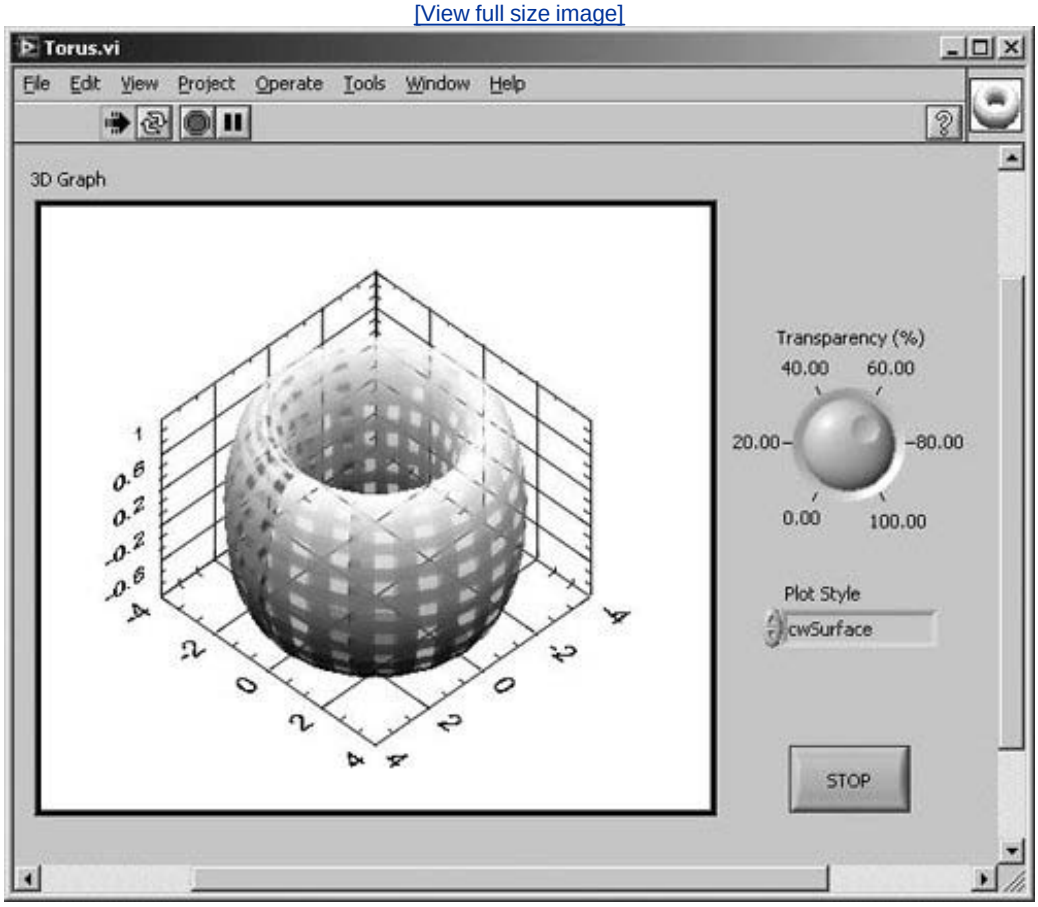
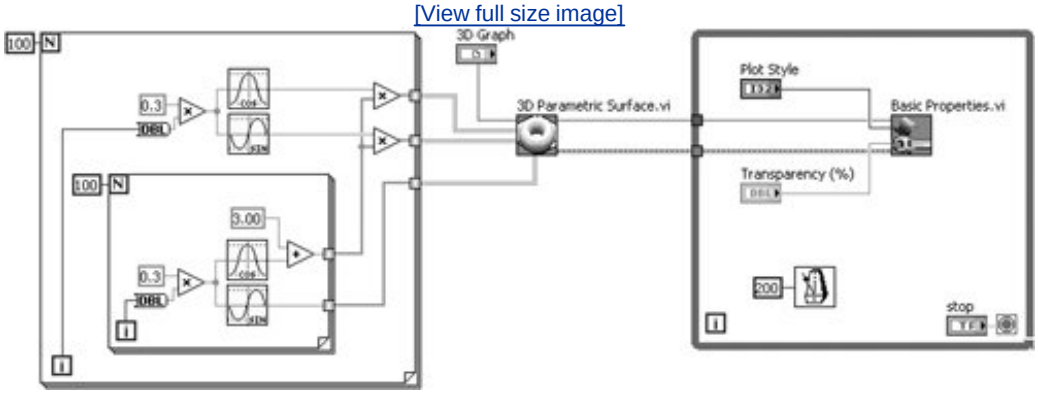


Figure 8.55. Torus.vi block diagram

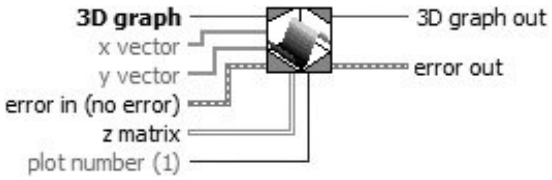
[View full size image]



The three types of 3D graphs can all plot in three dimensions, but each one works slightly differently:

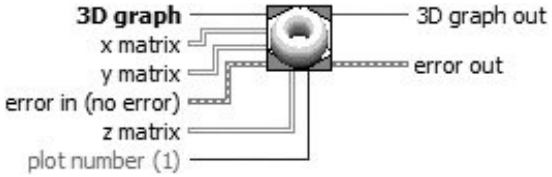
A 3D Surface Graph plots a simple surface from the z matrix (see Figure 8.56). The surface is modified by the x and y vectors that cause the surface to shift in relation to the x and y planes. It accepts one 2D array and the two optional 1D arrays.

Figure 8.56. 3D Surface Graph



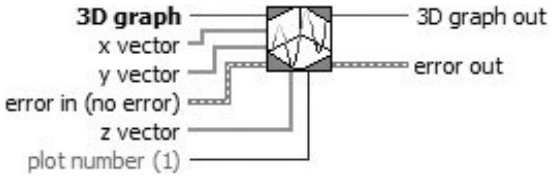
3D Parametric Surface plots a surface in terms of x, y, and z surfaces (see Figure 8.57). It takes in three 2D arrays or matrices that specify each of the x, y, and z planes.

Figure 8.57. 3D Parametric Surface



3D Curve describes a line in terms of x, y, and z points (see Figure 8.58). This VI has three 1D arrays or vectors inputs that specify each point in the plot.

Figure 8.58. 3D Curve



3D graphs can be more complicated than the simple charts and graphs you've worked withthey are really an advanced topic, so we won't say more about them here; we just wanted to let you know they are there if you need them. You can browse LabVIEW's 3D graph examples to get an idea of

how to use them.

These 3D graphs are ActiveX controls and contain more functionality than is exposed by the 3D graph VIs. As you will learn in Chapter 16, "Connectivity in LabVIEW," you can use property and invoke nodes to access these additional properties and methods.

PREV NEXT

Time Stamps, Waveforms, and Dynamic Data

Often, data that you want to analyze or acquire is a function of time. For example, we might be interested in seeing how temperature varies over the time of day, or how vibrational waveforms look when plotted over a time axis.

LabVIEW has some special data types that make it easier for the casual user to analyze and present this type of data in plots. These are the time stamp, waveform, and dynamic data. Time stamps are used to store the timing information in waveforms and multiple waveforms can be stored in dynamic data. Because there is a natural dependency from time stamp to waveform to dynamic data, we will introduce these topics in this order.

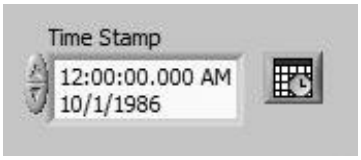
Time Stamp

The *time stamp* is a data type that stores an absolute date/time value, such as the time of a data collection event, with very high precision (19 digits of precision each in both the whole second and fractions of a second).

A numeric control can also be used to store and display time stamp values (we can change the display format to date/time), but the numeric control holds only a relative quantity. The time stamp control holds an absolute quantity.

In the previous activity, we used the Get Date/Time In Seconds function to obtain a time stamp of the current time (see Figure 8.59), which was used to set the t0 (initial time) of the sine waveform. The waveform data type uses a time stamp to store its t0 value.

Figure 8.59. Time Stamp control



The time stamp is not only a highly precise way of storing absolute time information; the time stamp control (shown in Figure 8.59) is very useful for viewing and editing time stamp values. The time stamp control may be found in the Modern>>Numeric subpalette of the Controls palette.

You can edit the time stamp value by clicking on the portion of time you wish to change and then using the up arrow and down arrow keys to increment and decrement the value. Or you can use your keyboard to type a value to replace the selected value. You can also pop up on the time stamp control and select Data Operations>>Set Time to Now to set the time stamp value to the current date and time.

Date/Time Browse Button

But, there is another way to edit the time stamp that is a lot more fun. Click the Time/Date Browse button to display the Set Time and Date dialog box, shown in Figure 8.60. From this dialog, you can easily edit the date and time value of the time stamp using a calendar-like interface.

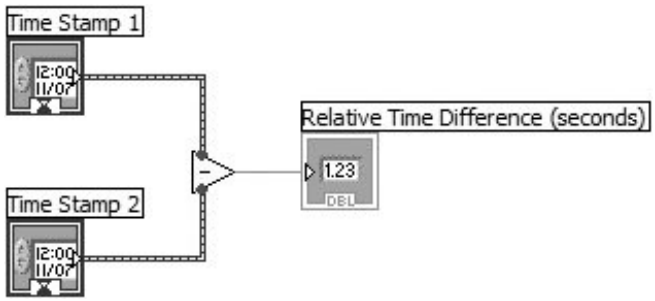
Figure 8.60. Set Time and Date dialog

You can also right-click time stamp controls, indicators, and constants and select Data Operations>>Set Time and Date from the shortcut menu to display the Set Time and Date dialog box. This is especially useful for time stamp constants and indicators, which do not have a Time/Date Browse button.

Relative Time Calculations

Often you will want to perform time calculations. For example, the code shown in Figure 8.61 shows how to calculate the relative time between two time stamps using the Subtract function.

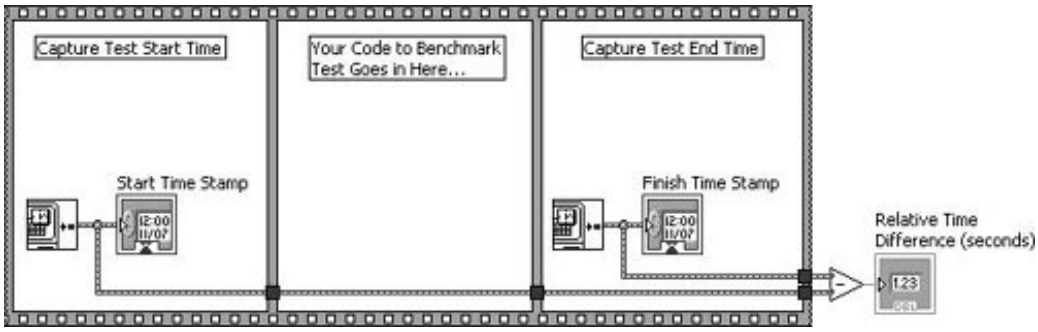
Figure 8.61. Calculating relative time by subtracting time stamps



For example, you can use this technique to build a performance benchmarking template, such as the one shown in Figure 8.62. Simply capture time stamps in the first and last frames of a Flat Sequence Structure and subtract the end time from the start time to yield the total execution time of your code in the middle frame. This is an excellent demonstration of how you can use a Flat Sequence Structure for application timing and synchronization tasks.

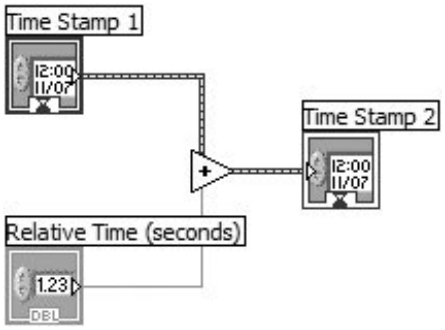
Figure 8.62. Performance benchmarking template that calculates the time required to execute the code in the center frame of a Flat Sequence Structure

[View full size image]



The operation of adding relative time to a time stamp can be done with the Add function, as shown in Figure 8.63.

Figure 8.63. Adding relative time to a time stamp

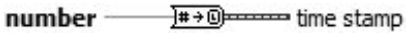


Time Stamp and Numeric Conversion

As you see in Figures 8.62 and Figure 8.63, the time stamp and numeric data types are closely related, and there are instances where you will need to convert between the two. Functions such as Add and Subtract will either adapt (because they are *polymorphic*) to a time stamp or coerce it to a DBL. But, sometimes you will want to explicitly perform this conversion. Use the To Double Precision Float function (available from the Programming>>Numeric>>Conversion palette) to convert a time stamp to a DBL (see Figure 8.64). Use To Time Stamp (also available from the Programming>>Numeric>>Conversion palette, but more easily available from the Programming>>Timing palette) to convert a numeric to a time stamp (see Figure 8.65).

Figure 8.64. To Double Precision Float

Figure 8.65. To Time Stamp



The To Double Precision Float function (shown in Figure 8.64) will convert any numeric to a DBL. It also works for converting a time stamp to a DBL, which is a sometimes overlooked (but extremely useful) fact.

Waveforms

In many engineering and scientific applications, much of data you handle is a collection of values that vary over time. For example, audio signals are pressure values versus time; an EKG is voltage versus time; the variations in the surface of a liquid as a pebble drops in are x,y,z coordinates versus time; the digital signals used by computers are binary patterns versus time. LabVIEW provides you with a convenient way to organize and work with this kind of time-varying datathe *waveform* data type. A waveform data type allows you to store not only the main values of your data, but also the time stamp of when the first point was collected, the time delay between each data point, and notes about the data. It is similar to other LabVIEW datatypes like arrays and clusters; you can add, subtract, and perform many other operations directly on the waveforms. You can create Waveform and Digital Waveform controls on the front panel from the I/O palette (see Figure 8.66 and Figure 8.68). The corresponding block diagram representations are brown/orange for waveform terminal and green for the digital waveform terminal (see Figure 8.67 and Figure 8.69).

Figure 8.66. Waveform control

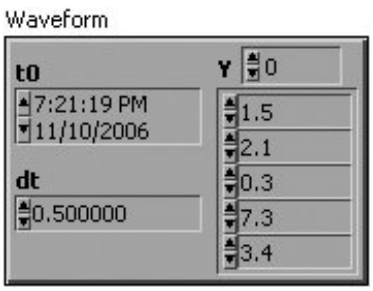


Figure 8.67. Waveform terminal

The analog waveform was introduced in LabVIEW prior to the introduction of the digital waveform, and it was simply called a "waveform." Thus, LabVIEW will still occasionally use "waveform" when referring to analog waveforms. In this book, we will try to use "analog waveform" and "digital waveform," but when referring to LabVIEW features, we will use the names used by LabVIEW. For example LabVIEW refers to the analog waveform control as "waveform" and the digital waveform control as "digital waveform."

Figure 8.68. Digital Waveform control

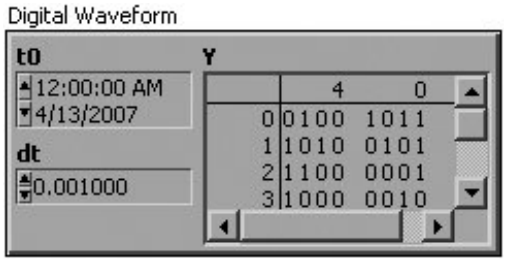


Figure 8.69. Digital Waveform terminal

Analog Waveform Symbol

LabVIEW uses different symbols for analog and digital waveforms, as you can see on the terminals shown here. The analog waveform symbol looks like a small sine wave and the digital waveform symbol looks like a small square wave. You will see these same symbols on waveform functions, VI icons, and elsewhere. So keep a sharp eye out for themthey will help you identify which waveform data type you are working with.

Digital Waveform Symbol

Also, the color of the analog waveform type is brown with orange (the colors of clusters and floating points), while the color of the digital waveform type is green (the color of Booleans). The wire pattern of all waveforms is the same as that of the cluster.

Examining the waveform data type a little more closely, we see it is really just a special type of cluster that consists of four components, called Y, t0, dt, and Attributes:

- Y: This component is the data that varies with time. For an analog waveform, this is a 1-D array of numeric data points, which can be either a single point or another waveform, depending on the operation. The representation of the analog 1-D array is DBL. Discrete signals (signals that usually only have two states, such as true or false, 0 or 5V, on or off, etc.) are called "digital" signals. For these, we use a digital waveform. For a digital waveform, the *Y* component is *digital data*, which is another special data type that we will discuss shortlyfor now you can think of it as a table of binary values where the columns are the digital lines and the rows are the successive valuesbasically, a 1-D array of port states (a port is a group of digital lines).
- t0: This component is a time stamp value that represents the time (according to the system clock) when the first point in the Y array was acquired. It is also referred to as the initial time or the time stamp. We learned about the time stamp data type in the previous section.
- dt: t, or delta-t, is a scalar value that represents the time between data points in the Y array.
- Attribute: By default, this component is hidden (you can see it by popping up and selecting Visible Items>>Attribute). Attributes, which are always optional, are variant data types that conveniently allow you to bundle custom information (as named attributes of the variant) along with your waveform such as the device number or channel number of your data acquisition system. Attributes do not affect how the data looks or its Y or time values. Variants are a very flexible data type that will accept any other LabVIEW data typewe will learn more about them in Chapter 12.

In Figure 8.66, the waveform control shows that the first point of the waveform starts at 07:21:19 PM, on Nov. 10, 2006, and that the time between each point is 0.5 seconds.

Waveforms Versus Arrays

In many ways, you can think of waveforms as just 1-D arrays that hold your datawith some extra information attached about the time and timing of the data points. Waveforms are most often useful

in analog data acquisition, which we'll discuss in Chapter 10, "Signal Measurement and Generation: Data Acquisition," and Chapter 11, "Data Acquisition in LabVIEW."

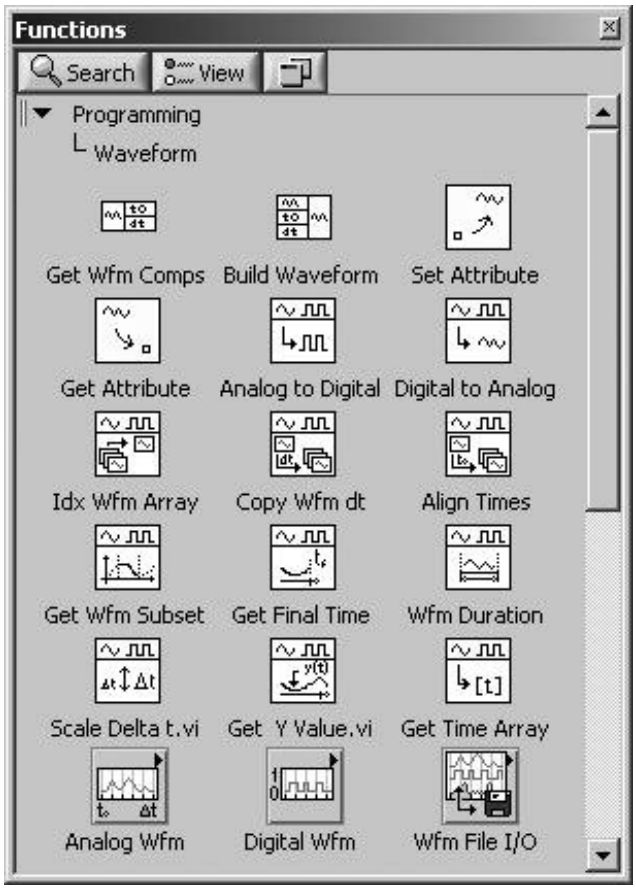
Of course, you do not necessarily need to use waveform data types; you can always just use arrays to hold your data. However, waveform data types offer several advantages over arrays:

- The presence of t0. Without t0, you would not know when your data were taken. The waveform data type automatically returns the time of day and the date in the t0 component, which gives you a real-world acquisition time for your data.
- Easier Graphing. The waveform data type also simplifies graphing your data. In previous versions of LabVIEW, you had to bundle the value of the initial point (x0) and the time between points (delta x) with your data (Y array). The waveform data type already contains these elements, so all you have to do is wire it to the graph.
- Easier Multiple Plot Graphing. The waveform data type also simplifies multiple plot graphs. In previous versions of LabVIEW, you had to bundle your x0, delta x, and Y array for each plot, and then send them to a build array to get a multiple plot graph. Using a waveform data type, you just wire a 1-D array of waveforms to the graph for a multiple plot. If, for example, you are acquiring data on multiple channels with an analog input VI, the VI automatically returns a 1-D array, so all you do is wire it directly to the graph.

Waveform Functions

In the Functions palette, you'll find an entire subpalette beneath the Programming category dedicated to the waveform manipulation, aptly named Waveform (see Figure 8.70).

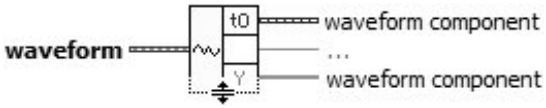
Figure 8.70. The Waveform palette



The Get Waveform Components and Build Waveform functions are used to get and set components of the analog waveform, digital waveform, and digital data.

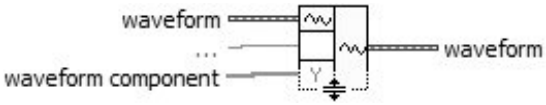
Returns the waveform components you specify. You specify components by right-clicking and selecting Add Element and creating an indicator. This function is expandable.

Figure 8.71. Get Waveform Components



Builds a waveform or modifies an existing waveform. If you do not wire an input to waveform, Build Waveform creates a new waveform based on the components you enter. If you do wire an input in waveform, the waveform is modified based on the components you specify. This function is expandable.

Figure 8.72. Build Waveform

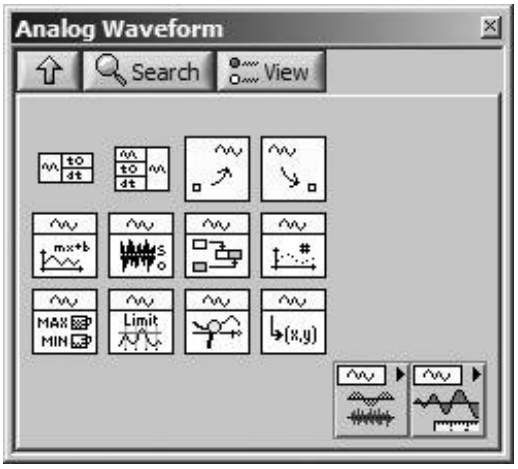


These two functions are polymorphic and can operate on analog waveforms, digital waveforms, and digital data. The waveform symbol on these functions will change depending on the data type wired to the waveform input. The analog waveform symbol looks like a small sine wave and the digital waveform symbol looks like a small square wave. You will see these same symbols on waveform functions, VI icons, and elsewhere. So keep a sharp eye out for themthey will help you identify which waveform data type you are working with.

The Waveform palette also has subpalettes with many useful functions and operations you can perform on waveforms.

The VIs in the Analog Waveform palette are used to perform arithmetic and comparison functions on waveforms, such as adding, subtracting, multiplying, finding the max and min points, concatenating, and so on (see Figure 8.73). Note that in most waveform operations that involve two or more waveforms, the waveforms involved must all have the same dt values.

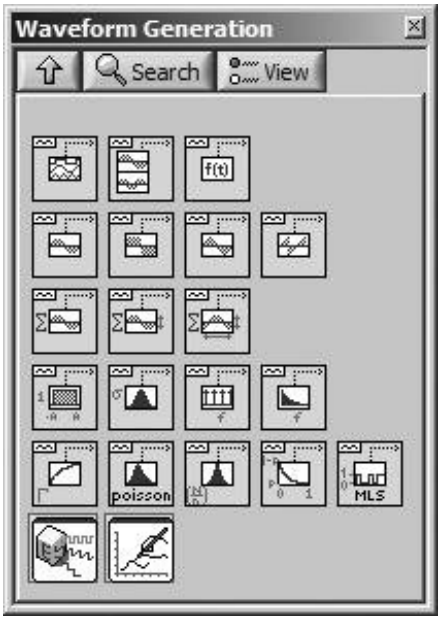
Figure 8.73. Analog Waveform palette



Analog Waveform Symbol

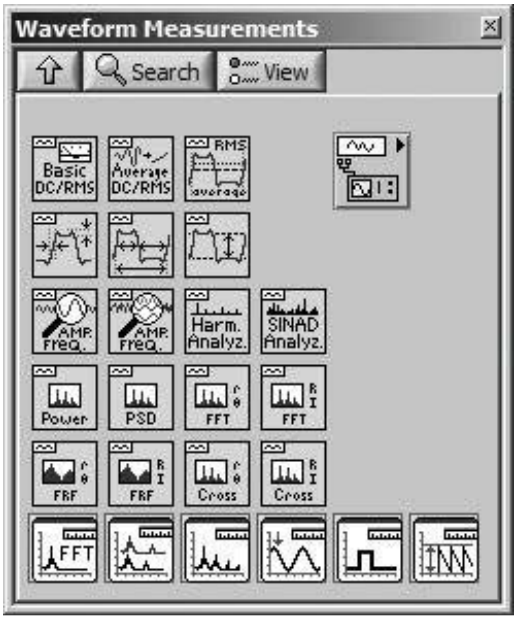
The Analog Waveform>>Waveform Generation palette allows you to generate different types of single and multitone signals, function generator signals, and noise signals (see Figure 8.74). For example, you can generate a sine wave, specifying the amplitude, frequency, and so on.

Figure 8.74. Waveform Generation palette



The Analog Waveform>>Waveform Measurements palette allows you to perform common time and frequency domain measurements such as DC, RMS, Tone Frequency/Amplitude/Phase, Harmonic Distortion, SINAD, and Averaged FFT measurements (see Figure 8.75).

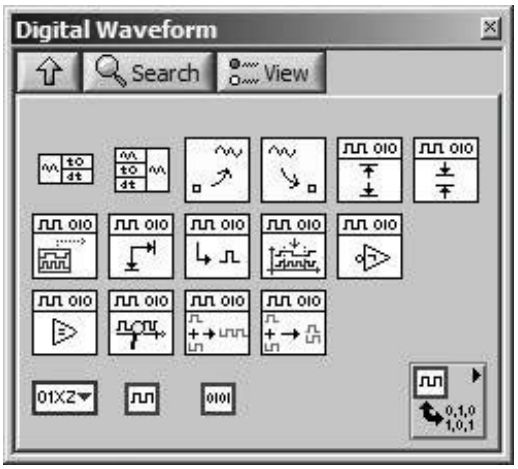
Figure 8.75. Waveform Measurements palette



Digital Waveform Symbol

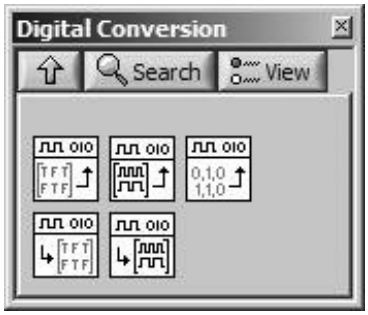
The Digital Waveform palette allows you to perform operations on digital waveforms and digital data, such as searching for digital patterns, compressing and uncompressing digital signals, and so on (see Figure 8.76).

Figure 8.76. Digital Waveform palette



The Digital Waveform>>Digital Conversion palette allows you to convert to and from digital data (see Figure 8.77).

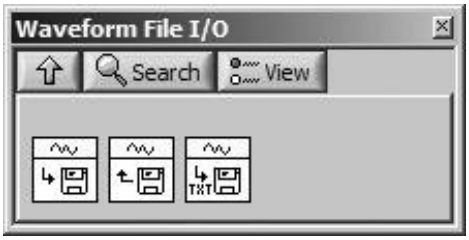
Figure 8.77. Digital Conversion palette



The functions in Waveform File I/O allow you to write waveform data to and read waveform data

from files (see Figure 8.78).

Figure 8.78. Waveform File I/O palette



One last thing you should know about waveformswhen you need to plot an analog waveform, you can wire it directly to a waveform chart or a waveform graph. The terminal automatically adapts to accept a waveform data type and will reflect the timing information on the X axis. Digital waveforms have their own digital waveform graph, which we will discuss later.

Let's look at a simple example of how to use and plot a waveform in the next activity.

Activity 8-6: Generate and Plot a Waveform

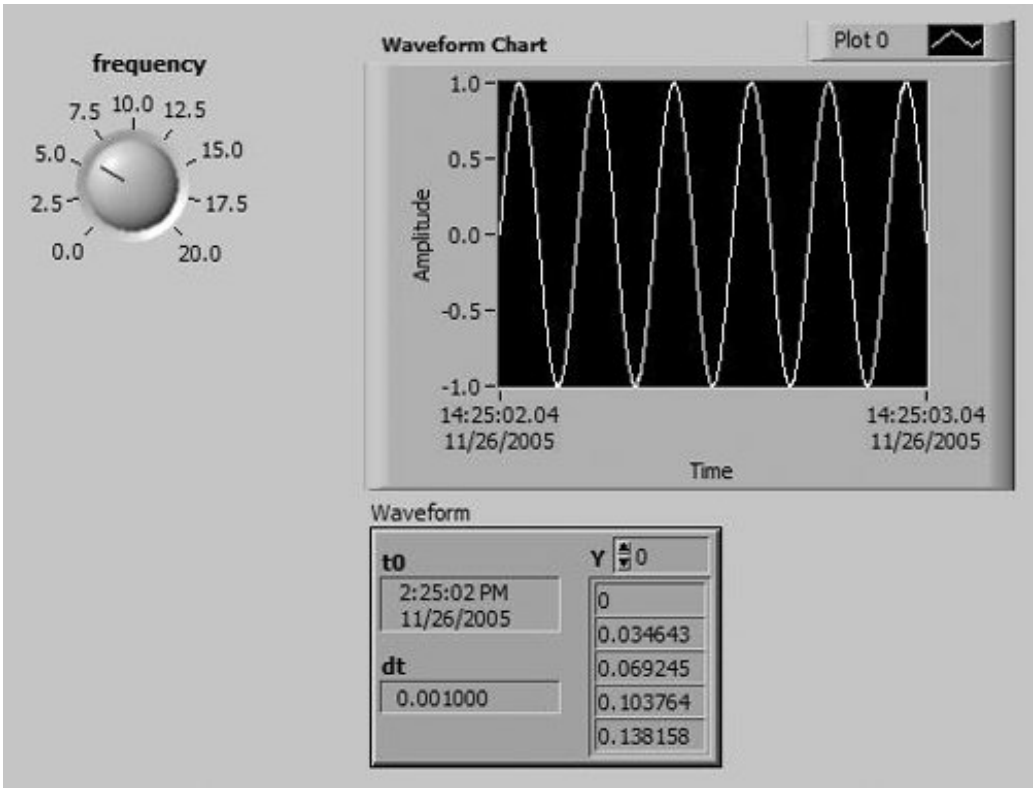
In this activity, you will generate a sine waveform, set its initial time stamp to the current time, and plot it on a chart.

1. Open a new front panel.

2. On the front panel, place a dial (from the Numeric palette), a chart, and a waveform indicator. Label the dial frequency, change its maximum value, and set it to 1 digit of precision (from the Format and Precision pop-up menu option), so that it looks like Figure 8.79.

Figure 8.79. Front panel of the VI you will create during this activity

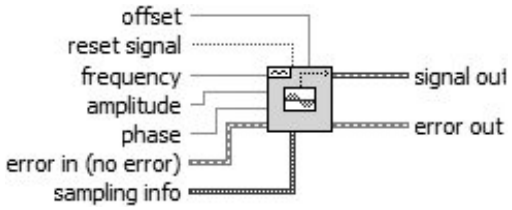
[View full size image]



3. Build a block diagram that will generate a sine waveform and plot it. The waveform generation functions do not provide a value for the t0 component of the waveform (they return the default time stamp value of 00:00:00.000 PM, MM/DD/YYYY, which, when converted to a DBL floating point in absolute time display mode, is midnight GMT 1/1/1904 (7:00:00 PM EST 12/31/1903), so use the Build Waveform to set the t0 to the current time. Here are the functions you will use the following:

   Sine Waveform.vi generates a waveform containing a sine wave with a null time stamp (00:00:00.000 PM, MM/DD/YYYY, or midnight GMT 1/1/1904 if represented as a DBL floating point in absolute time display mode). See Figure 8.80.

Figure 8.80. Sine Waveform.vi

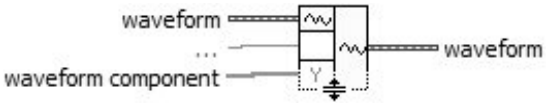


The Get Date/Time in Seconds returns the current system date and time, as a time stamp data type (see Figure 8.81).

Figure 8.81. Get Date/Time in Seconds

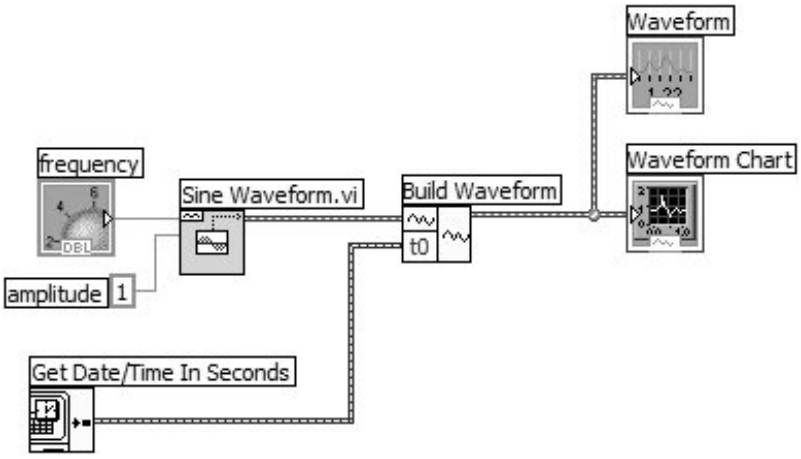
This function allows you to modify components of the waveform; in this activity, just t0 (see Figure 8.82).

Figure 8.82. Build waveform



Your block diagram should look like Figure 8.83.

Figure 8.83. Block diagram of the VI you will create during this activity



4. Run and test your VI at different frequencies. You may want to pop up on the chart and select X Scale>>Autoscale X and Y Scale>>Autoscale Y. In addition, clearing the chart (by selecting Data Operations>>Clear Chart from its pop-up menu) between runs will make it more intelligible.

5. Save your VI as `Waveform Exercise.vi`.

We'll take another look at waveforms again in Chapter 11, where the analog waveform data type is used in some of the analog input data acquisition functions.

Digital Data

Digital waveforms have a special data type for the *Y* component: a table of binary states (binary values can be either 0 or 1)the columns in the table are the digital lines and the rows are the successive values.

You can create a Digital Data control on the front panel from the I/O palette (Figure 8.84). The corresponding block diagram representation is a green terminal (Figure 8.85).

Figure 8.84. Digital data control

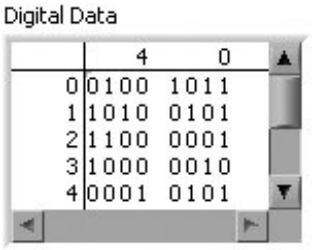
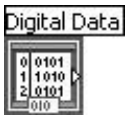


Figure 8.85. Digital data terminal



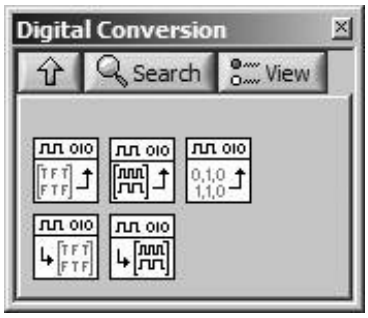
0101

Digital Data Symbol

Note the digital data symbol on the digital data terminal, which looks like binary data (0101). This symbol is also found on functions and VIs that operate on digital data.

To easily convert to and from digital data, use the VIs located on the Programming>>Waveform>>Digital Waveform>>Digital Conversion palette (shown in Figure 8.86). These VIs allow you to convert digital data to and from Boolean arrays and Integer arrays, as well as import digital data from a spreadsheet string.

Figure 8.86. Digital Conversion palette



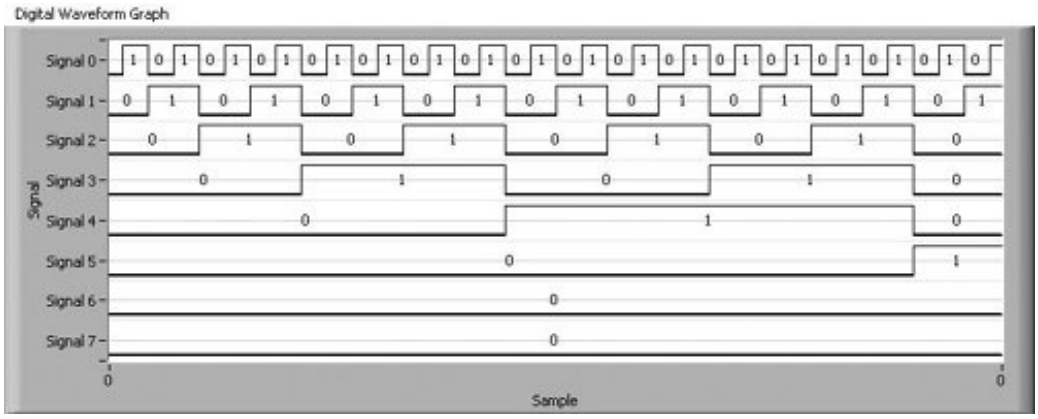
You can also operate on digital data using the Get Waveform Components and Build Waveform functions (which are also used to operate on waveforms), but this is not recommended.

Digital Waveform Graphs

For displaying digital waveform data, use the digital waveform graph (from the Modern>>Graph), as shown in Figure 8.87. If you've worked with digital logic before, the digital waveform graph isn't hard to use; otherwise, don't even worry about learning it! You can find some good examples on using digital waveform graphs in LabVIEW's built-in examples (`examples\general\graphs\DWDT Graphs.llb`).

Figure 8.87. Digital waveform graph

[View full size image]



Dynamic Data

Nearly all of the Express VIs for acquiring, analyzing, manipulating, and generating signals use a special data type, called the dynamic data type, for passing signal data. Dynamic data is simply one or more channels of waveform datain fact, you can think of dynamic data as simply an array of analog waveforms, wrapped in a very smart wire. However, dynamic data is *very* smart, in that it makes it very easy for you to perform operations like merging signals into a single wire. For example, you can wire dynamic data directly to other dynamic data, and LabVIEW will automatically insert a *Merge Signals* function to combine the two signals into a single wire, as shown in Figure 8.88 and Figure 8.89.

Figure 8.88. Wiring dynamic data to an existing dynamic data wire (before)

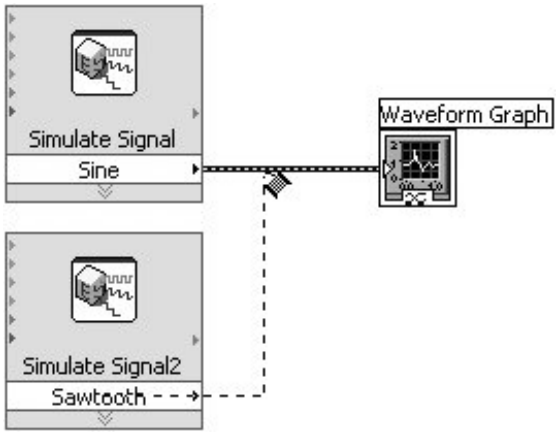
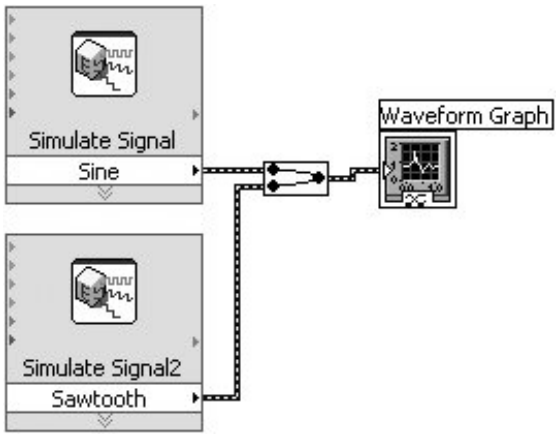
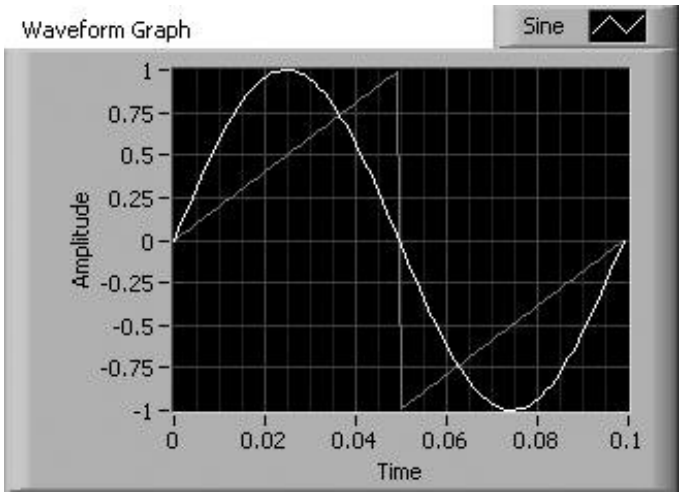


Figure 8.89. Wiring dynamic data to an existing dynamic data wire (after)



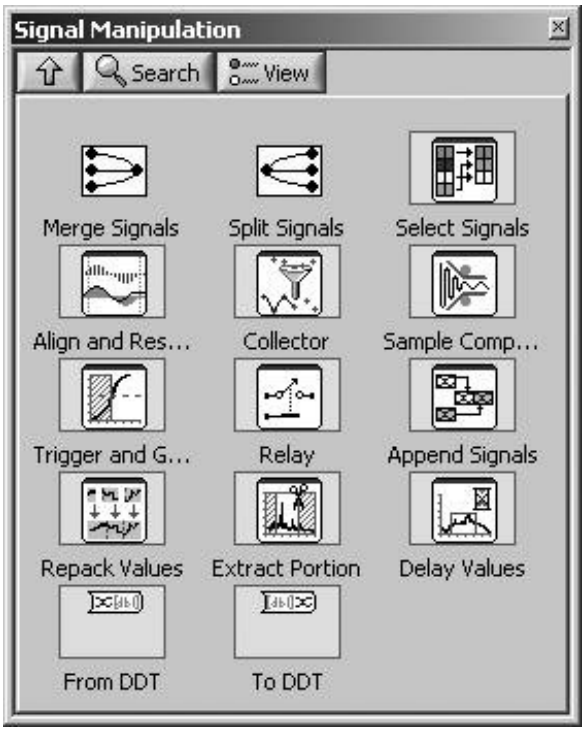
If we look at a graph of our merged signal, shown in Figure 8.90, you can see that the resulting merged signal contains both a Sine signal and a Sawtooth signal.

Figure 8.90. Waveform graph displaying two dynamic data plots



The basic functions for manipulating dynamic data are found on the Express>>Signal Manipulation palette, shown in Figure 8.91. Here you will find functions for merging, splitting, and selecting signals, as well as the To DDT and From DDT functions, which are used to convert to and from traditional LabVIEW data types like waveforms, arrays, and scalars.

Figure 8.91. Signal Manipulation palette



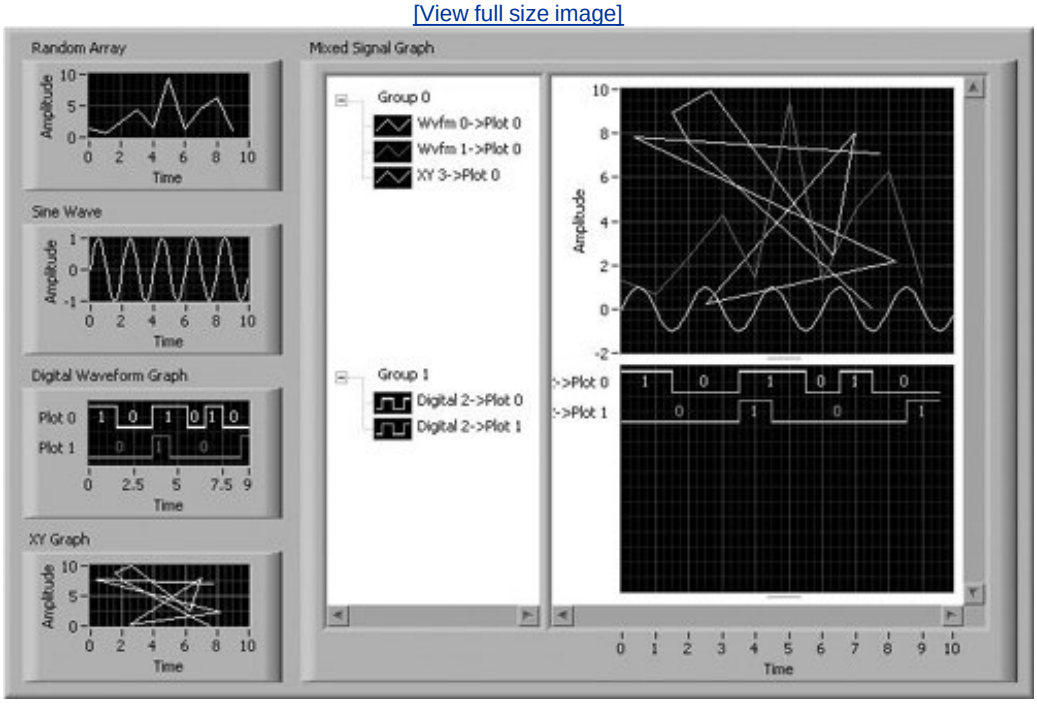
PREV NEXT

Mixed Signal Graphs

When you have both analog and digital signals that you wish to display together in a way that shows the timing relationships between the two, use a mixed signal graph, which may be found on the Modern>>Graph palette of the Controls palette (see Figure 8.92).

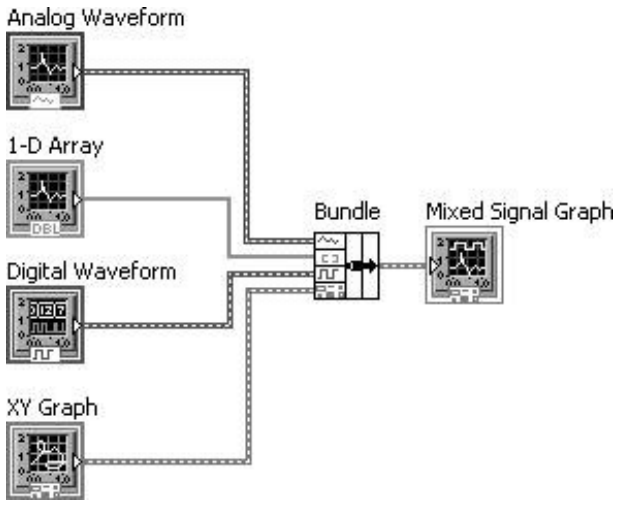
Figure 8.92. Mixed signal graph

[View full size image]



This graph accepts a cluster of any elements that you could wire into the waveform graph, XY graph, or digital graph, as shown in Figure 8.93.

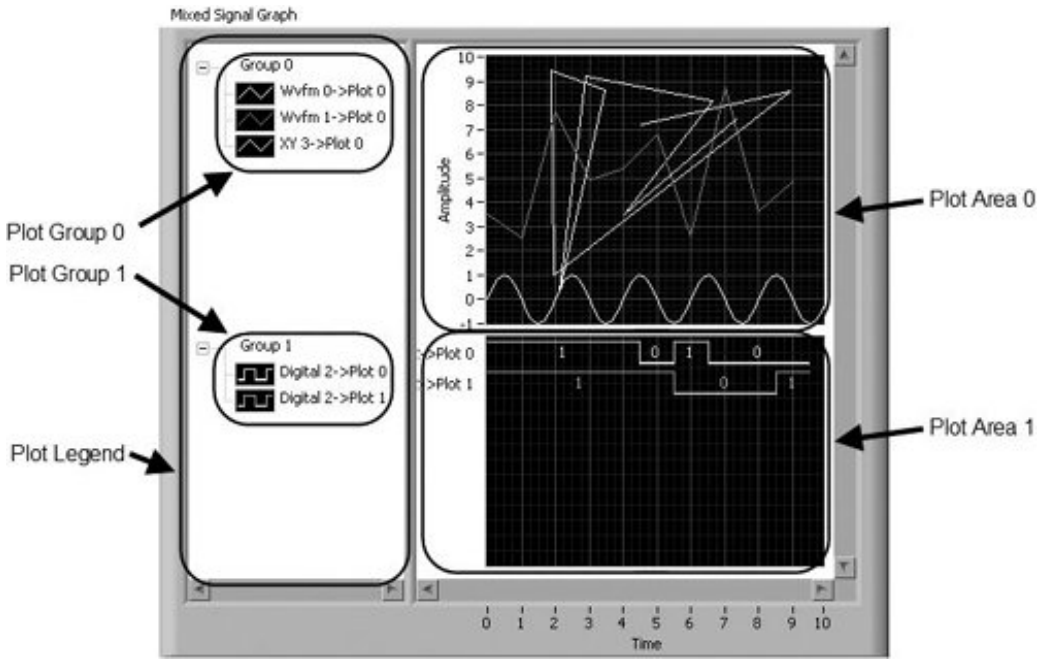
Figure 8.93. Building a mixed signal graph by bundling multiple graphable datatypes



It will plot all of these elements on one or more plot areas. You can add plot areas from the pop-up menu of an existing plot area by selecting Add Plot Area. Similarly, you can remove a plot area by selecting Remove Plot Area from the pop-up menu. The plot legend of a mixed signal graph is a tree control, having each plot's label and attribute display as child-nodes of the group name. You can drag and drop plots from one group to another; however, you cannot have analog and digital data in the same plot group. In fact, when you are wiring data on the block diagram to the mixed signal graph, LabVIEW will force you to have at least two plot areas if both analog and digital data are included (see Figure 8.94).

Figure 8.94. Mixed signal graph parts

[View full size image]



For an example of the mixed signal graph in action, see `examples\general\graphs\Mixed Signal Graph.vi`.

Multi-Plot Cursor

As we mentioned in the "Graph Cursors" section, the mixed signal graph has a special graph cursor mode called Multi-Plot mode (this cursor mode is available only to the mixed signal graph, and will be grayed out for other graph types). This type of graph cursor can report the *Y* values of multiple plots at a common *X* value. To create a multi-plot cursor, right-click on the cursor legend and select Create Cursor>>Multi-Plot, as shown in Figure 8.95. In order to report the *Y* values of a plot, the cursor must be configured to watch the plot, by selecting the plot name from the cursor label's pop-up menu under the Watch submenu. To watch all plots, select Watch>>All Plots from the cursor label's pop-up menu (see Figure 8.96).

Figure 8.95. Create a multi-plot cursor

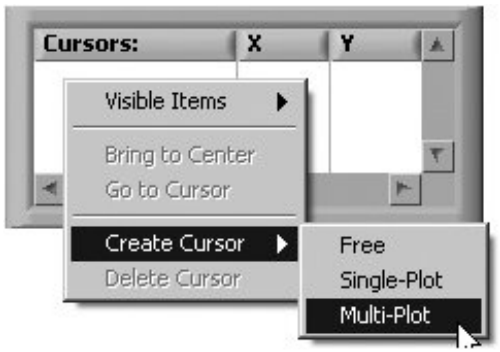
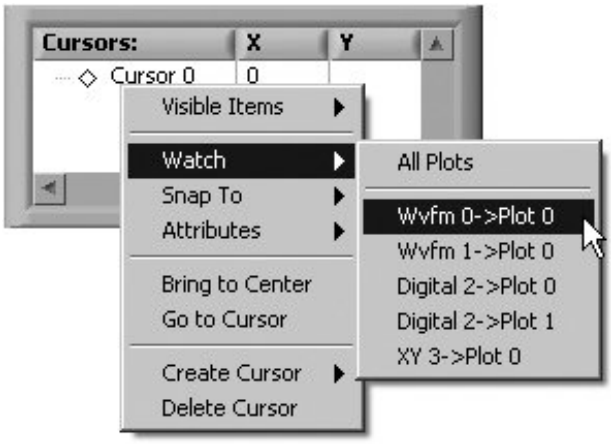


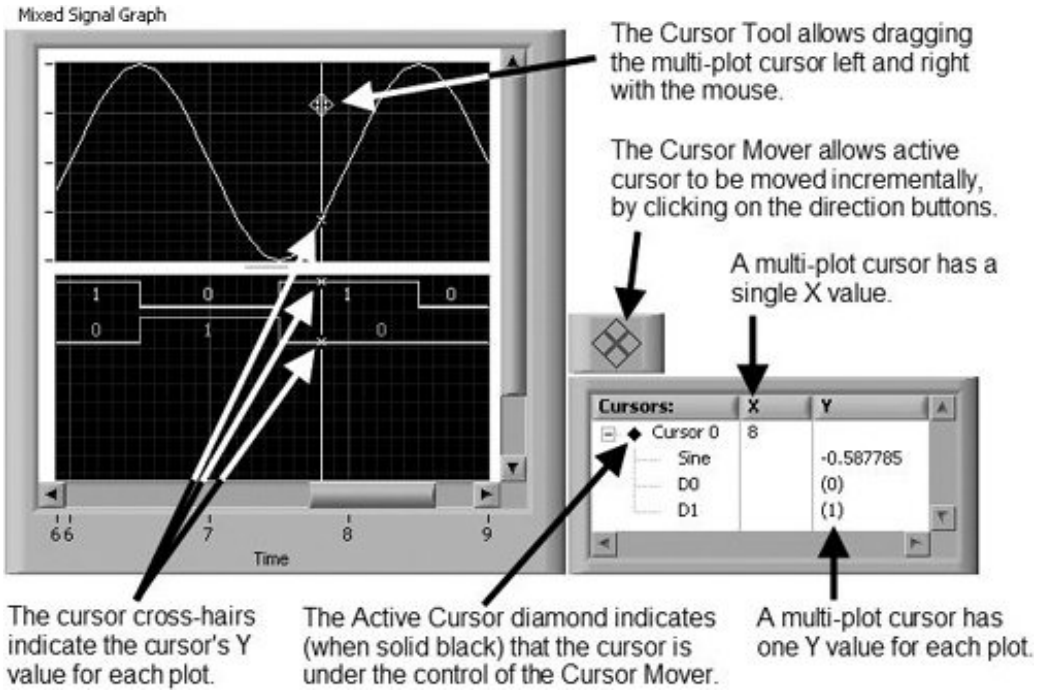
Figure 8.96. Watch a plot



Once you have created a multi-plot cursor, you can change its X value by grabbing the vertical cursor line and moving it horizontally (or by using the cursor mover). Watch how the cursor reports the Y values of each plot that it is watching, as shown in Figure 8.97.

Figure 8.97. Multi-plot cursor

[View full size image]

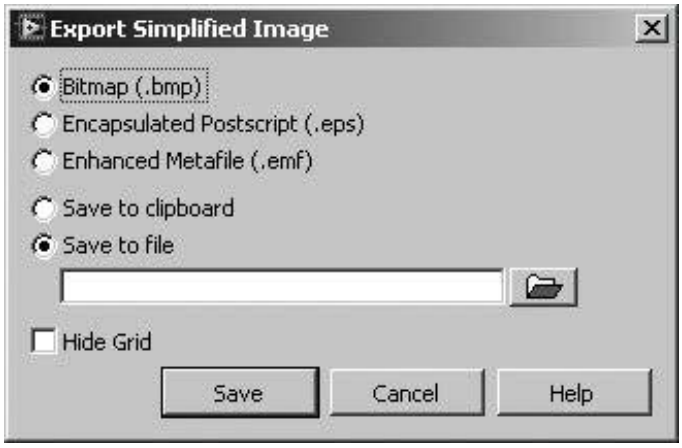

Mixed Signal Graph
The Cursor Tool allows dragging the multi-plot cursor left and right with the mouse.
The Cursor Mover allows active cursor to be moved incrementally, by clicking on the direction buttons.
A multi-plot cursor has a single X value.
Cursors:
X
Y
Cursor 0
8
Sine
-0.587785
D0
(0)
D1
(1)
66
7
8
9
Time
The cursor cross-hairs indicate the cursor's Y value for each plot.
The Active Cursor diamond indicates (when solid black) that the cursor is under the control of the Cursor Mover.
A multi-plot cursor has one Y value for each plot.

PREV NEXT

Exporting Images of Charts and Graphs

Often you will want an image of your graphed data for a report or presentation. LabVIEW makes this very easyjust right-click on your graph or chart and select Export Simplified Image from the shortcut menu (this option will appear beneath the Data Operations>> submenu when the VI is in edit mode). From the Export Simplified Image dialog (shown in Figure 8.98), you can choose to save the image to disk in one of several file formats, or save the image to the clipboard for pasting into another document.

Figure 8.98. Export Simplified Image dialog



The "simplified" image that LabVIEW exports contains only the plot area, digital display, plot legend, and index display and does not include scrollbars, the scale legend, the graph palette, or the cursor palette. You can choose whether to include the grid, by setting the value of the *Hide Grid* checkbox in the Export Simplified Image dialog.

The following file formats are supported:

- Windows .emf, .bmp, and .tif
- Mac OS X .pict, .bmp, and .tif
- Linux .bmp and .tif

The Export Simplified Image option is not available for intensity graphs or intensity charts.

PREV NEXT

Wrap It Up!

You can create exciting visual displays of data using LabVIEW's charts and graphs. *Charts* append new data to old data, interactively plotting one point (or one set of points) at a time, so you can see a current value in context with previous values. *Graphs*, on the other hand, display a full block of data after it has been generated. *Waveforms*, a new data type we learned about, can be used with both charts and graphs.

LabVIEW provides several kinds of graphs: *waveform* graphs, *XY* graphs, *intensity* graphs, *3D* graphs, and *digital waveform* graphs.

The *waveform graph* plots only single-valued points that are evenly distributed with respect to the independent variable, such as time-varying waveforms. In other words, a graph plots a Y array against a set timebase.

The *XY graph* is a general-purpose, Cartesian graph that lets you plot multivalued functions such as circular shapes. It plots a Y array against an X array.

Intensity plots are excellent for displaying patterned data because they can plot three variables of data against each other on a 2D display. Intensity charts and graphs use color to represent the third variable. They accept a 2D array of numbers, where each number is mapped to a color and the number's indices in the array specify location for the color on the graph or chart. In most other ways, intensity plots function like standard two-variable charts and graphs.

3D graphs (Windows only) are more sophisticated and perspective-oriented 3D graphs that allow you to plot (x, y, z) coordinates in 3D space. A variety of functions is provided in LabVIEW for manipulating and configuring 3D graphs.

The *digital waveform graph* is a special type of graph used for plotting digital time-domain data. It is particularly useful for showing true/false states changing over time.

The mixed signal graph can display almost any type of data accepted by other graphs. Just bundle the plots together into a cluster and wire them into the mixed signal graph. A multi-plot cursor is used on a mixed signal graph to show the timing relationships between multiple plots.

You can configure the appearance of charts and graphs using the plot legend, the scale legend, and the graph palette. You can also change the scales to suit your data and bring up cursors to mark your plots.

Both charts and graphs can draw multiple plots at a time. Data types can get tricky, so you may want to refer to the examples in this chapter or those that ship in the examples directory as templates while writing your own graphing VIs.

Mechanical action of Boolean switches allows you to control how they behave when you click on them. You can set a switch to return to its default value after its new value has been read oncethat way, it's all ready to be used again. This type of action is called latch action. You can also specify if you want the mouse click to register when you press the mouse button or release it.

The time stamp is a data type in LabVIEW that stores an absolute date/time value with very high precision (19 digits of precision each in both the whole second and fractions of a second). Use the subtract function to calculate relative time (as a DBL) from time stamps, or the add function to add relative time (as a DBL) to a time stamp.

A waveform is a special LabVIEW data type that stores information about the initial time stamp and time interval between data in a series of data points. There is an entire Waveforms palette on the Functions palette that provides you with all sorts of functions to perform on waveforms. You can wire waveform data directly to a chart or graph to plot it.

The dynamic data type is a very "smart" type of data that contains one or more waveforms. You can use dynamic data types with Express VIs. Dynamic data is simply one or more channels of waveform data, wrapped in a very smart wire. It makes it very easy for you to perform operations like merging signals into a single wire.

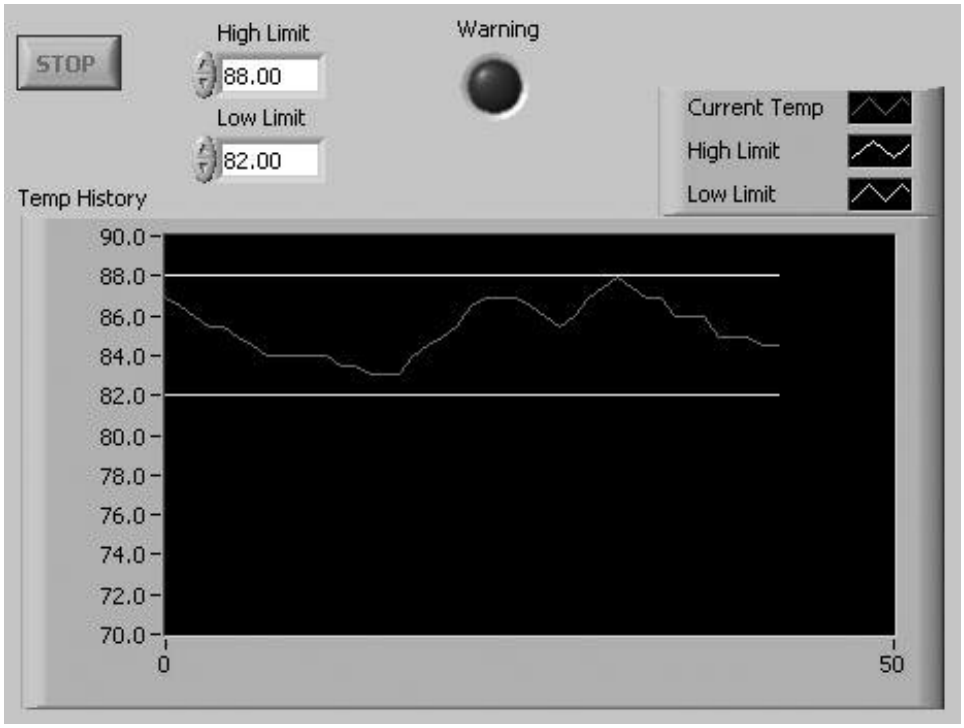
PREV NEXT

Additional Activities

Activity 8-7: Temperature Limit

Build a VI that continuously measures the temperature once per second and displays the temperature on a chart in scope mode. If the temperature goes above or below the preset limits, the VI turns on a front panel LED. The chart should plot the temperature, as well as the upper and lower temperature limits. You should be able to set the limits from the front panel. Take a look at the front panel shown in Figure 8.99 for a start. Name the VI Temperature Limit.vi.

Figure 8.99. Temperature Limit.vi front panel



Activity 8-8: Max/Min Temperature Limit

Modify the VI you created in Activity 8-7 to display the maximum and minimum values of the temperature trace. Name the VI Temp Limit (max-min).vi.

You must use shift registers and the Array Max & Min function (Array palette).

Activity 8-9: Plotting Random Arrays

Build a VI that generates a 2D array (three rows by ten columns) containing random numbers. After generating the array, index off each row and plot each row on its own graph. Your front panel should contain three graphs. Name the VI Extract 2D Array.vi.

PREV NEXT

9. Exploring Strings and File I/O

Overview

This chapter introduces some of the powerful things you can do with strings. LabVIEW has many built-in string functions, similar to its array functions, which let you manipulate string data for screen display, for sending and receiving commands and data from instruments, or any number of reasons. You will also learn how to save data to and retrieve data from a disk file, including how to use the configuration file VIs to read and write to INI files. You will learn what the term "spreadsheet file" really means and what you need to know to allow users to open them in their external spreadsheet applications. You will also learn how regular expressions can help you easily find any pattern, such as email addresses, within a string.

Goals

- Learn more about options for string controls and indicators
- Understand how to use LabVIEW's string functions
- Convert numeric data to string data, and vice versa
- Use the file input and output (I/O) VIs to save data to a disk file and then read it back into LabVIEW
- Use the configuration file VIs to save and recall key-value pairs to and from configuration (INI) files

Key Terms

- Scrollbar
- Listbox
- Table
- "\" codes display
- Spreadsheet file
- Comma Separated Values (CSV) file
- Text file
- Binary file
- Configuration (INI) file
- Formatting string
- Regular expression

More About Strings

We introduced strings in Chapter 4, "LabView Foundations"a string is simply a collection of ASCII characters.[1] Often, you may use strings for more than simple text messages. For example, in instrument control, you pass numeric data as character strings. You then convert these strings to numbers to process the data. Storing numeric data to disk can also use strings; in many of the file I/O VIs, LabVIEW first converts numeric values to string data before it saves them to a file.

[1] At the time of this writing, the current release of LabVIEW, 8.0, does not support Unicode, although it's a feature that may appear soon.

Choose Your Own Display Type

String controls and indicators have several options you might find useful. For example, they can display and accept characters that are normally nondisplayable, such as backspaces, carriage returns, and tabs. If you choose "\" Codes Display (instead of Normal Display) from a string's pop-up menu, nondisplayable characters appear as a backslash (\) followed by the hexadecimal value of the ASCII character.

Table 9.1 shows the meaning of the various "\" codes.

Table 9.1. LabVIEW "\" Codes

Code	*LabVIEW Implementation*
\00 \FF	Hexadecimal value of an 8-bit character; alphabetical characters must be uppercase
\b	Backspace (ASCII BS, equivalent to \08)
\f	Formfeed (ASCII FF, equivalent to \0C)
\n	New line (ASCII LF, equivalent to \0A)
\r	Return (ASCII CR, equivalent to \0D)
\t	Tab (ASCII HT, equivalent to \09)
\s	Space (equivalent to \20)
\\	Backslash (ASCII \, equivalent to \5C)

You should know that some commonly used non-printable characters, such as space (0x20), tab (0x09), carriage return (0x0D), new line (0x0A), and so on are not shown in "\" codes display with a

hexadecimal value following the backslash (\). Instead these common non-printable characters have a single *lowercase* letter that represents the non-printable character. For example, you will never see "\20" (0x20 is the hexadecimal value of the space character), only "\s" (the "\" code of the space character). In fact, if you type "\20" into a string that is configured for "\" codes display, the "\20" will immediately be converted to "\s."

You must use uppercase letters for hexadecimal characters and lowercase letters for the special characters, such as formfeed and backspace. LabVIEW interprets the sequence `\BFare` *as hex BF followed by the word "are," whereas LabVIEW interprets* `\bFare` *and* `\bfare` *as a backspace followed by the words "Fare" and "fare." In the sequence* `\Bfare`, `\B` *is not the backspace code, and* `\Bf` *is not a valid hex code. In a case like this, when a backslash is followed by only part of a valid hex character, LabVIEW assumes a zero follows the backslash, so LabVIEW interprets* `\B` *as hex 0B. Any time a backslash is not followed by a valid character code, LabVIEW ignores the backslash character.*

This string shows the Backslash code for the New Line character in Normal Display mode (see Figure 9.1).

Figure 9.1. String control in Normal Display mode

After switching to "\" Codes Display mode, the string shows the spaces entered by the user (see Figure 9.2).

Figure 9.2. String control in '\' Codes Display mode

Don't worrythe data in the string does not change when the display mode is toggled; only the display of certain characters changes. '\' Codes Display mode is very useful for debugging programs and for specifying nondisplayable characters required by instruments, the serial port, and other devices.

Strings also have a Password Display option, which sets the string control or indicator to display a "*" for every character entered into it, so that no one can see what you type. Although the front panel shows only a stream of "****," the block diagram reads the actual data in the string. Obviously, this display can be useful if you need to programmatically password-protect all or part of your VIs. Figures 9.3 and 9.4 show the front panel and block diagram of a VI that passes data from a string control in Password Display mode to a string indicator in Normal Display mode. Note that the string data remains unchanged, regardless of the display mode.

Figure 9.3. String control in Password Display mode (left) and string indicator in Normal Display mode (right) displaying the same string data

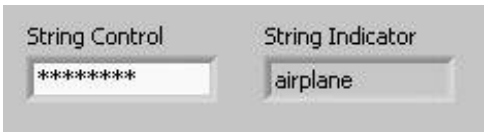
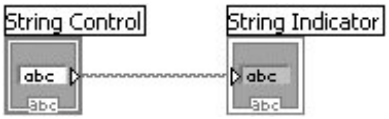


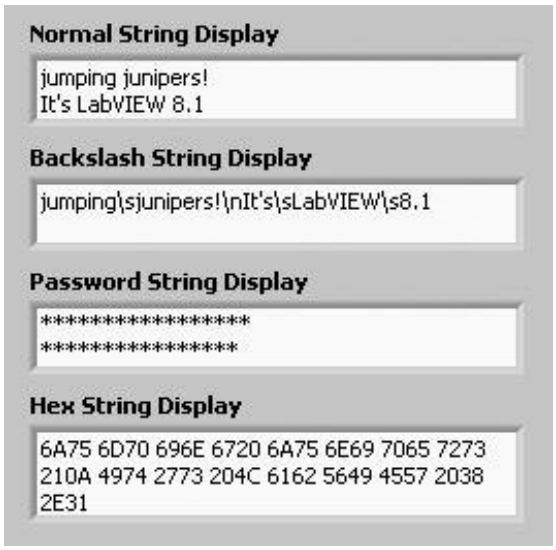
Figure 9.4. Block diagram showing that the string control passes its data, unchanged, to the string indicator



If you want to see your string as hexadecimal characters instead of alphanumeric characters, use the Hex Display option.

Figure 9.5 shows the same string displayed in each of the four display options: Normal, "\", Password, and Hex.

Figure 9.5. The same string data displayed in each of the four display modes



Single Line Strings

When a string control is used as a text entry box on a user interface VI, it is often desirable to limit the string to a single line. In this scenario, when a user presses <enter> or <return>, it signifies that she is finished entering text. If you desire this single line behavior, select the Limit to Single Line option from a string's pop-up menu. If this option is not enabled, then the string is not limited to a single line: hitting <return> causes the cursor to jump to a new line to let you type more.

Updating While You Type

Normally, string controls don't change their value on their terminal in the block diagram until you finish typing and either hit <enter>, click outside the string box, or click on the "✓" button on the VI toolbar to indicate the string entry is complete. Most of the time you will want this behavior, because you don't want your block diagram code to evaluate an incomplete string before a user is finished typing.

If you do want the value to be updated as you type (just like a knob, for example), pop up on the string control and select Update Value While Typing.

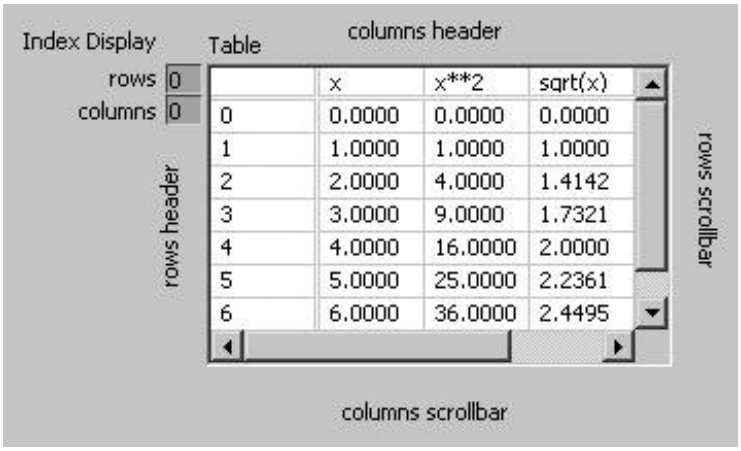
The Scrollbar

If you choose the Visible Items>>Scrollbar option from the string pop-up Visible Items>> submenu, a vertical scrollbar appears on the string control or indicator. You can use this option to minimize the space taken up on the front panel by string controls that contain a large amount of text. Note that this option will be grayed out unless you've increased the size of your string enough for a scrollbar to fit.

Tables

A *table* is a special structure that displays a two-dimensional (2D) array of strings. You can find it in the Lists & Table subpalette of the Controls palette. A table in all its glory is shown in Figure 9.6.

Figure 9.6. Table control



Tables have row and column headings that you can show or hide; the headings are separated from the data space by a thin open border space. You can enter text headings using the Labeling tool or Operating tool (like everything else). You can also update or read headings using property nodes, which you've heard so much about and will learn how to use soon enough.

Like an array index display, a table index display indicates which cell is visible at the upper-left corner of the table.

For a good example of a table and how to use it, open and run Building Tables.vi, located in `EVERYONE\CH09`.

Listboxes

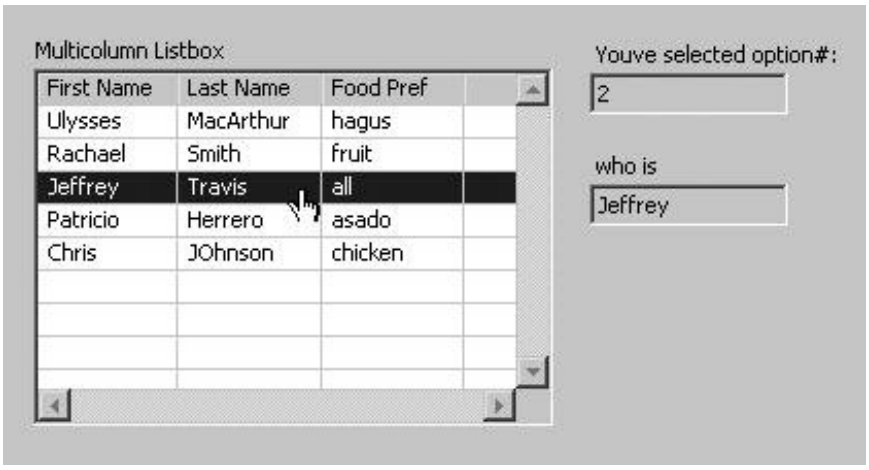
There are two kinds of listboxes in LabVIEW: Listbox and Multi-column Listbox, from the List & Table palette.

A listbox control looks similar to a table, but it behaves very differently at run-time. During edit mode, you can type any text into a listbox just like you would a table. When you run the VI, the listbox acts as a "multiple choice" menu where you can click on any row to highlight it and select it.

A listbox's value is the row number that is selected (or an array of row numbers, if the listbox is configured to allow multiple row selection via the Selection Mode>>0 or More Items or Selection Mode>>1 or More Items pop-up menu options). This is very different from a table control, whose value is a 2D array of strings. For a listbox, you can programmatically change the item strings that appear in the control by writing to the Item Names property (using a property node, which you will learn about in Chapter 13, "Advanced LabVIEW Structures and Functions").

Figure 9.7 shows a multi-column listbox.

Figure 9.7. Multi-column listbox



Listboxes are useful when you want to present data (either single-column or multiple-columns) to a user, but you just want them to choose one or more of the items.

If text does not fit in a cell of a table or listbox, a tip strip with the full text appears when you run the VI and place the cursor of the Operating tool over the cell.

Pop up on a single-column listbox and select Multi-line Input to allow cell entries to contain multiple lines.

You can select Autosize Row Height from the pop-up menu of a single-column listbox to configure how the rows change height for font changes and multiple-line text entries.

To allow users to edit the cell string values of a listbox, at run-time, pop up on a listbox and select Editable Cells.

For a simple example of how a listbox works, see Listbox Example.vi in EVERYONE\CH09.

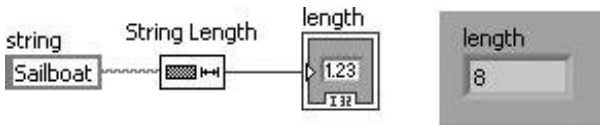
PREV NEXT

Using String Functions

Like arrays, strings can be much more useful when you take advantage of the many built-in functions provided by LabVIEW. This section examines a few of the functions from the String subpalette of the Functions palette. You might also browse through the rest of this palette to see what other functions are built in.

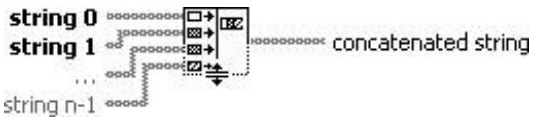
String Length returns the number of characters in a given string (see Figure 9.8).

Figure 9.8. String Length



Concatenate Strings concatenates all input strings into a single output string (see Figure 9.9).

Figure 9.9. Concatenate Strings



When first placed onto the block diagram, this function has two inputs. You can resize it with the Positioning tool (as shown in Figure 9.10) to increase the number of inputs.

Figure 9.10. Using the Concatenate Strings function to build a sentence from its parts

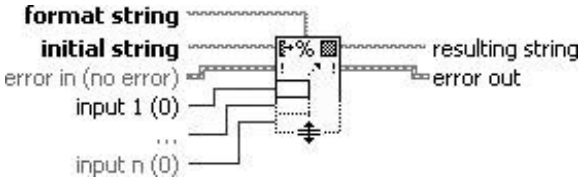
[View full size image]

In addition to simple strings, you can also wire a one-dimensional (1D) array of strings as input; the output will be a single string containing a concatenation of strings in the array (see Figure 9.11).

Figure 9.11. Using the Concatenate Strings function to build a sentence from a string and an array of strings

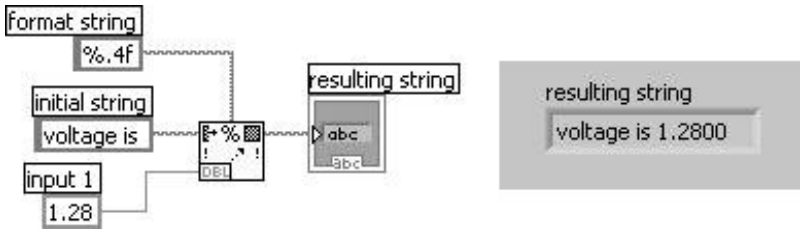
In many instances, you must convert strings to numbers or numbers to strings. The Format Into String and Scan From String functions have these capabilities (as do various other functions, but we'll concentrate on these). We'll talk about Format Into String now and Scan From String in a little while.

Figure 9.12. Format Into String



Simply put, Format Into String formats string, path, enumerated type, time stamp, Boolean, or numeric data as text. Figure 9.13 shows Format Into String converting the floating-point number 1.28 to the 6-byte string "1.2800."

Figure 9.13. Using the Format Into String function to format numeric floating point data



Format Into String isn't just for converting numeric data to string data (although that is how it is very commonly used). You can also convert from strings, paths, enums, Booleans, and time stamps. For example, using the format string %s will convert a Boolean to the string "FALSE" or "TRUE." However, using the format string %d will convert it to the string "0" or "1." Similarly, using the format string %s will convert an enum into its text value, and using the format string %d will convert it into its integer value.

Format Into String formats the input argument (which is in numeric format) as a string, according to the format specifications in format string. These specifications are listed in detail in the LabVIEW manuals and Online Reference (search on "String Function Overview"). The function appends the newly converted string to the input wired to initial string, if there is one, and outputs the results in resulting string. The following table gives some examples of Format Into String's behavior. Table 9.2 shows the resulting string outputs for various combinations of inputs to Format Into String.

Table 9.2. Resulting String Outputs for Given Format Into String Inputs

Initial String	*Format String*	*Number*	*Resulting String*
(empty)	score=%2d%%	87	score=87%
score=	%2d%%	87	score=87%
(empty)	level=%7.2eV	0.03642	level=3.64E-2V
(empty)	%5.3f	5.67 N	5.670 N

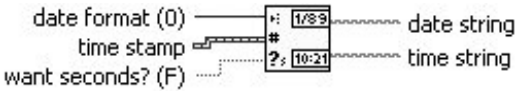
The "%" character begins the formatting specification. Given *"%number1.number2*," *number 1* specifies the field width of the resulting string and *number 2* specifies the precision (i.e., number of digits after the decimal point). An "f" formats the input number as a floating-point number with fractional format, "d" formats it as a decimal integer, and "e" formats it as a floating-point number with scientific notation.

Format Into String can be resized to convert multiple values to a single string simultaneously.

You can use localization codes in your format string (which are not displayed, but affect the output of all subsequent entries in the format specifier string) to specify the decimal separator for numeric output. Use "%,;" for comma decimal separator, "%.;" for period decimal separator, and "%;" for system default decimal separator.

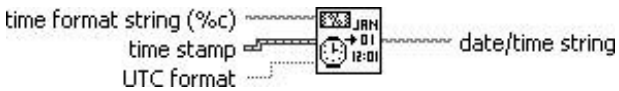
Get Date/Time String (found in the Programming>>Timing palette) outputs date string, which contains the date specified by the time stamp input, and time string, which contains the time specified by the time stamp input (see Figure 9.14). This function is useful for time stamping your data. Note that if you don't wire the *time stamp* input to Get Date/Time String; it will use the current time.

Figure 9.14. Get Date/Time String



Format Date/Time String (found in the Programming>>Timing palette) formats a time stamp value or a numeric value as time in the format you specify using *time format codes* in the time format string input (see Figure 9.15). This function is much more powerful than the Get Date/Time String function for creating data/time strings.

Figure 9.15. Format Date/Time String



The *time format codes* used by Format Date/Time String are shown in Table 9.3.

Table 9.3. Time Format Codes for the Format Date/Time String Function

Time Format Code	*Meaning*
%a	abbreviated weekday name (e.g., "Sat")
%b	abbreviated month name (e.g., "Feb")
%c	locale-specific date/time
%d	Two-digit day of month
%H	hour, 24-hour clock
%I	hour, 12-hour clock
%m	Two-digit month number (e.g., "03")
%M	Minute
%p	a.m./p.m. flag (e.g., "AM" or "PM")
%S	Second
%x	locale-specific date
%X	locale-specific time
%y	year within century (e.g., "08")
%Y	year including century (e.g., "2008")
%<digit>u (e.g. %3u)	fractional seconds with <digit> precision (e.g., ".456")

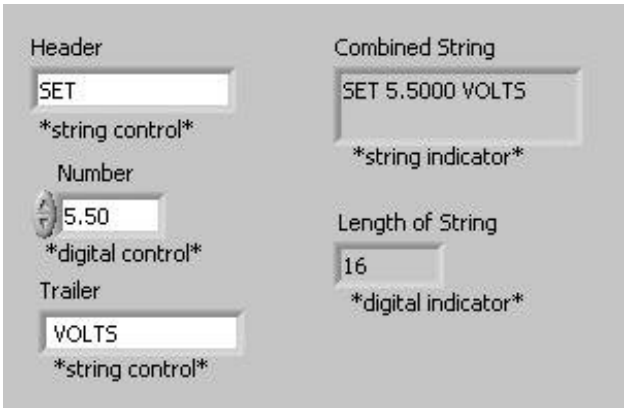
For example, using the format string "%Y-%m-%d %H:%M:%S%3u" might produce a `date/time string` output that looks something like "2006-03-31 15:22:58.296." (Note how the %<digit>u time format code adds a decimal point in front of the fractional seconds.)

Activity 9-1: String Construction

It's time for you to practice and give a new meaning to the phrase "strings attached." You will build a VI that converts a number to a string and concatenates that string with other strings to form a single output string. The VI also determines the length of the output string.

1. Build the front panel shown in Figure 9.16.

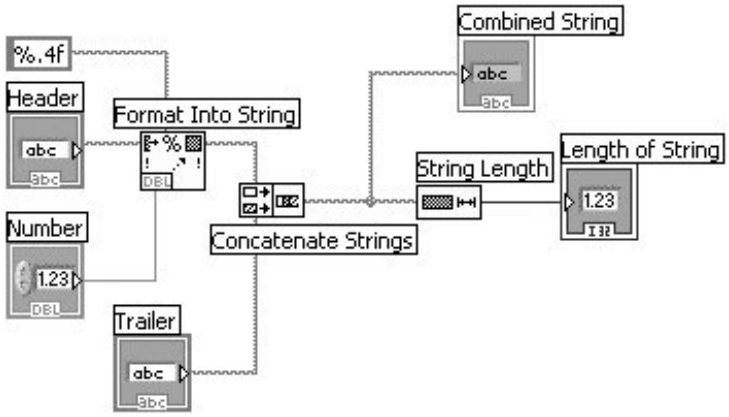
Figure 9.16. Front panel of the VI you will create during this activity



The VI will concatenate the input from the two string controls and the digital control into a single output string, which is displayed in the string indicator. The digital indicator will display the string's length.

2. Build the block diagram pictured in Figure 9.17.

Figure 9.17. Block diagram of the VI you will create during this activity



Format Into String Function

Format Into String (Programming>>String palette) converts the number you specify in the `Number` digital control to a string with fractional format and four digits of precision.

Concatenate Strings Function

Concatenate Strings function (Programming>>String palette) combines all input strings into a single output string. To increase the number of inputs, stretch the icon using the Positioning tool.

String Length Function

String Length function (Programming>>String palette) returns the number of characters in the concatenated string.

3. Return to the front panel and type text inside the two string controls and a number inside the digital control. Make sure to add spaces at the end of the header and the beginning of the trailer strings, or your output string will run together. Run the VI.

4. Save and close the VI. Name it Build String.vi and place it in your `MYWORK` directory. Do you feel like a LabVIEW expert yet? You're getting there!

PREV NEXT

Parsing Functions

Sometimes you will find it useful to take strings apart or convert them into numbers, and these parsing functions can help you accomplish these tasks.

String Subset accesses a particular section of a string. It returns the substring beginning at offset and containing length number of characters. Remember, the first character's offset is zero (see Figure 9.18). Figure 9.19 shows an example of how String Subset can be used to return a subset of an input string.

Figure 9.18. String Subset

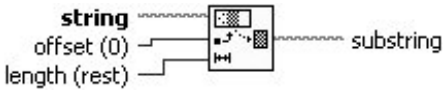
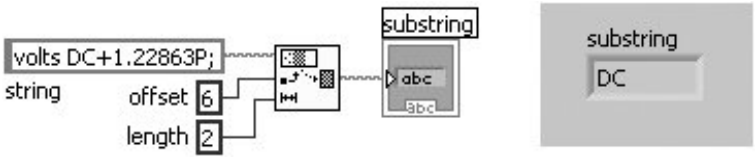
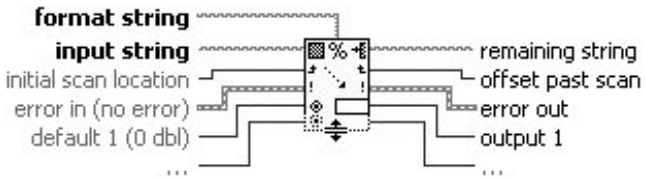


Figure 9.19. String Subset used to return a subset of an input string



Scan From String, the "opposite" of Format Into String, converts a string containing valid numeric characters (0 to 9, +, -, e, E, and period) to numeric data (see Figure 9.20). This function starts scanning the input string at initial search location and converts the data according to the specifications in format string (to learn more about the specifications, see the LabVIEW manuals or "String Function Overview" in the Online Reference). Scan From String can be resized to convert multiple values simultaneously.

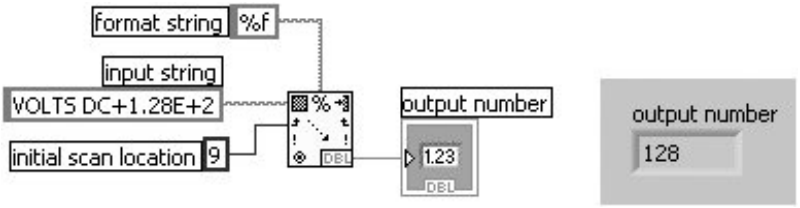
Figure 9.20. Scan From String



a, b, c, d, e, f, A, B, C, D, E, and F are valid characters if a hex format is specified, and comma may be valid if it is the localized decimal point.

In this example, Scan From String converts the string `"VOLTS DC + 1.28E + 2"` to the number 128.00 (see Figure 9.21). It starts scanning at the eighth character of the string (which is the + in this caseremember that the first character offset is zero).

Figure 9.21. Scan From String used to extract a floating point numeric from an input string



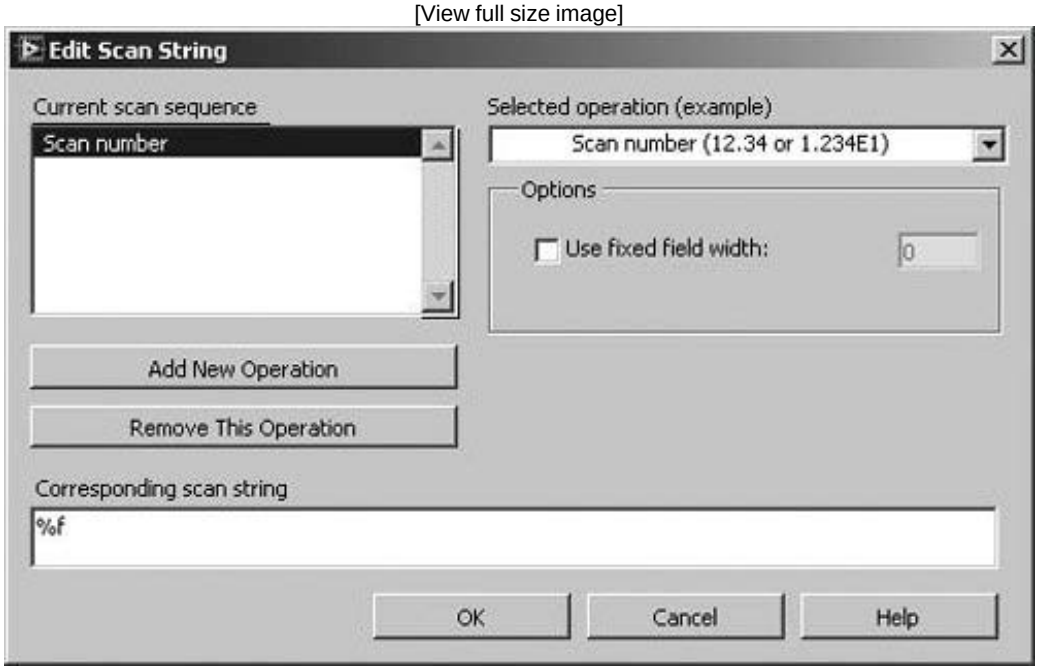
Scan From String is also capable of converting string data to more than just numeric data. You can extract strings, paths, enums, Booleans, and time stamps from strings. For example, using the format string %s will extract a "FALSE" or "TRUE" from a string, as a Boolean. Similarly, you can use the format string %d to extract a "0" or "1" from a string, as a Boolean. However, you must be careful with scanning non-numeric data from strings, as the Scan From String function will often stop scanning when it encounters a space (or

other whitespace) character. Therefore, it is not quite as flexible as Format Into String is for the inverse operation.

Both Format Into String and Scan From String have an Edit Scan String interface that you can use to create the format string. In this dialog box, you can specify format, precision, data type, and width of the converted value. Double click on the function or pop up on it and select Edit Format String to access the Edit Scan String or Edit Format String dialog box (see Figure 9.22).

Figure 9.22. Edit Scan String dialog

[View full size image]

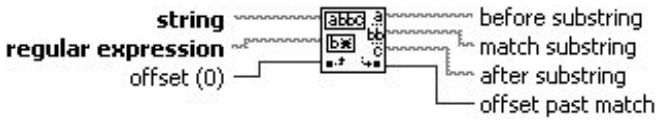


After you create the format string and click the Create String button, the dialog box creates the string constant and wires it to the format string input for you.

Match Pattern and Regular Expressions

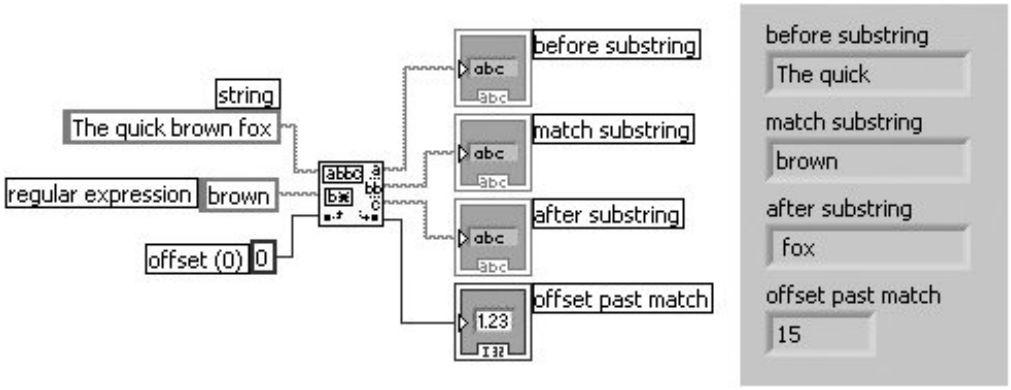
Match Pattern is used to look for a given pattern of characters in a string (see Figure 9.23). It searches for and returns a matched substring . Match Pattern looks for the *regular expression* (or *pattern*) in string , beginning at offset ; if it finds a match, it splits the string into three substrings. If no match is found, the match substring is empty and offset past match is set to -1.

Figure 9.23. Match Pattern



A regular expression is a string that uses a special syntax (called *regular expression syntax*) to describe a set of strings that match a pattern (see Figure 9.24)the syntax (and some useful examples) will be described shortly.

Figure 9.24. Match Pattern used to find a pattern in an input string



The Match Pattern function allows you to use some special characters to give you more powerful and flexible searches. Table 9.4 shows you the special characters you can use in the Match Pattern function.

.

Matches any character. For example, l.g matches lag, leg, log, and lug.

?

Matches zero or one instance of the expression preceding ?. For example, be?t matches bt and bet but not best.

\

Cancels the interpretation of any special character in this list. For example, \? matches a question mark and \. matches a period. You also can use the following constructions for the space and non-displayable characters:

\b

backspace

\f

form feed

\n

new line

\s

space

\r

carriage return

\t

tab

\xx

any character, where xx is the hex code using 0 through 9 and uppercase A through F

^

If ^ is the first character of regular expression, it anchors the match to the offset in string. The match fails unless regular expression matches that portion of string that begins with the character at offset. If ^ is not the first character, it is treated as a regular character.

[]

Encloses alternates. For example, [abc] matches a, b, or c. The following characters have special significance when used within the brackets in the following manner:

Indicates a range when used between digits, or lowercase or uppercase letters; for example, [025], [ag], or [LQ]. The following characters have significance only when they are the first character within the brackets.

~

Matches any character, including non-displayable characters, except for the characters or range of characters in brackets. For example, [~09] matches any character other than 0 through 9.

^

Matches any displayable character, including the space and tab characters, except the characters or range of characters enclosed in the brackets. For example, [^09] matches all displayable characters, including the space and tab characters, except 0 through 9.

+

Matches the longest number of instances of the expression preceding +; there must be at least one instance to constitute a match. For example, be+t matches bet and beet but not bt.

*

Matches the longest number of instances of the expression preceding * in regular expression , including zero instances. For example, be*t matches bt, bet, and beet.

$

If $ is the last character of regular expression , it anchors the match to the last element of string . The match fails unless regular expression matches up to and including the last character in the string. If $ is not last, it is treated as a regular character.

Table 9.4. Special Characters Used by the Match Pattern Function

Special Character	*Interpreted by the Match Pattern Function*

If you have used regular expressions before in other programming languages or with UNIX command-line utilities like `grep` , you know how powerful and useful regular expressions can be.

If you are not familiar with regular expressions, they are a very powerful set of rules for matching and parsing text. Table 9.5 shows you some examples of pattern matching using regular expressions. For more information on regular expression syntax, consult the LabVIEW documentation or a good reference.

VOLTS

All uppercase and lowercase versions of volts; that is, VOLTS, Volts, volts, and so on

[Vv][Oo][Ll][Tt][Ss]

A space, a plus sign, or a minus sign

[+-]

A sequence of one or more digits

[0-9]+

Zero or more spaces

\s* or * (that is, a space followed by an asterisk)

One or more spaces, tabs, new lines, or carriage returns

[\t \r \n \s]+

One or more characters other than digits

[~0-9]+

The word "Level" only if it begins at the offset position in the string

^Level

The word "Volts" only if it appears at the end of the string

Volts$

The longest string within parentheses

(.*)

The longest string within parentheses but not containing any parentheses within it

([~( )]*)

The character [

[[ ]

cat, dog, cot, dot, cog, and so on

[cd][ao][tg]

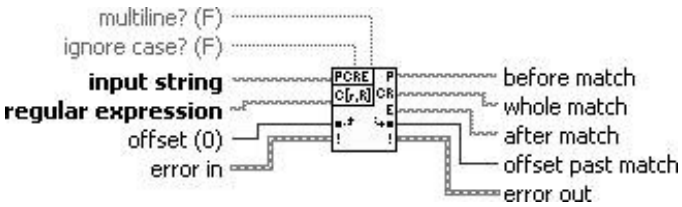
Table 9.5. Regular Expressions and the Characters That They Can be Used to Find

Characters to Find	*Regular Expression*

The Match Pattern is a relatively fast and powerful way to search for patterns in a string. It does not however, incorporate every aspect of the regular expression syntax. If you need more specialized options to match strings with regular expressions, you can use the Match Regular Expression function.

The Match Regular Expression function (see Figure 9.25) incorporates a larger set of options and special characters for string matching, but it is slower than Match Pattern. It uses the Perl Compatible Regular Expressions (PCRE) library, an open source library written by Philip Hazel at the University of Cambridge.

Figure 9.25. Match Regular Expression



Because of its additional complexity, Match Regular Expression should only be used if Match Pattern won't work for what you are trying to do.

Match Regular Expression is expandable. Resize it vertically (using the Positioning tool to drag the resize handles up or down) to show submatches.

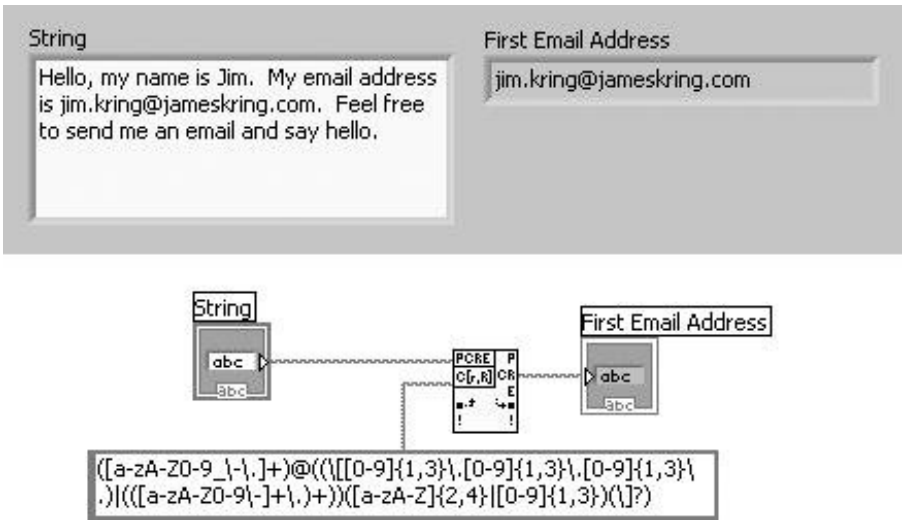
If you want to learn more about regular expressionsand there is a vast amount of information

availableuse your favorite search engine to find resources on the Web. There are even *regular expression libraries* (online databases of regular expressions) where you can find some very useful regular expressions. For example, here is a regular expression (we found on the Web) that will match an email address:

```
[View full width]([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z
([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)
```

Figure 9.26 shows this regular expression, in action. As you can see, regular expressions are an extremely powerful tool!

Figure 9.26. Using a regular expression to find email addresses



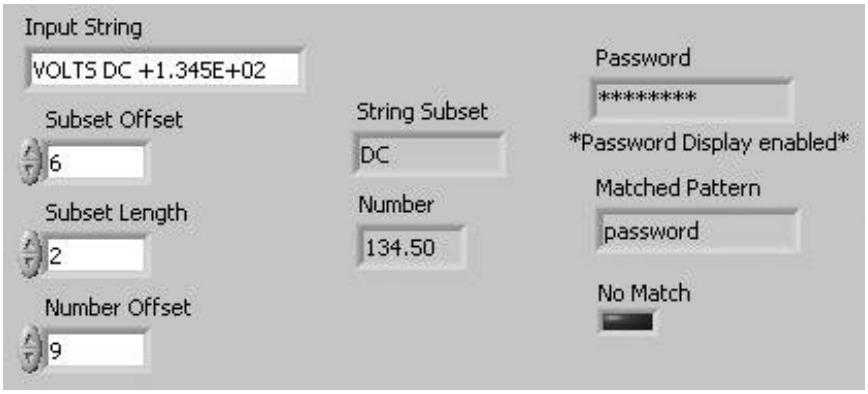
PREV
NEXT

Activity 9-2: More String Parsing

You will create a VI that parses information out of a longer string by taking a subset of a string and converting the numeric characters in that subset into a numeric value.

1. Build the front panel shown in Figure 9.27.

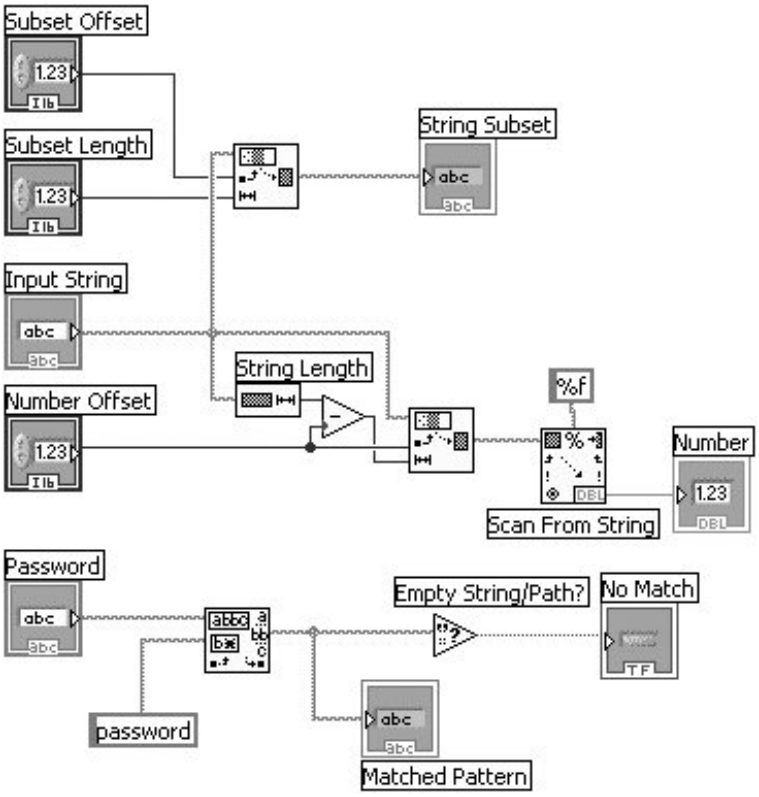
Figure 9.27. Front panel of the VI you will create during this activity



2. Set the `Password` string control to display only asterisks by selecting Password Display from its pop-up menu

3. Create the block diagram in Figure 9.28.

Figure 9.28. Block diagram of the VI you will create during this activity



String Subset Function

The String Subset function (Programming>>String palette) returns a subset of given length from the input string, according to the offset specified.

Scan From String Function

The Scan From String function (Programming>>String palette) converts a string containing valid numeric characters (0 to 9, +, -, E, and period [and sometimes a few others]) to a number.

Match Pattern Function

Match Pattern (Programming>>String palette) compares the user's input password string to a given password string. If there is a match, it is displayed; if not, the string indicator shows an empty string.

Empty String/Path? Function

Empty String/Path? (Programming>>Comparison palette) returns a Boolean TRUE if it detects an empty string from the match sub-string output of Match Pattern.

String Length Function

String Length function (Programming>>String palette) returns the number of characters in the concatenated string.

4. Run the VI with the inputs shown. Notice that the string subset of "DC" is picked out of 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 if you want; just remember that strings, like arrays, are indexed starting at zero.

   Also note how the Password string shows only "*****." Match Pattern checks the input password against a password string (which in this case contains the characters "password"), and then returns a match if it finds one. If it finds no match, it returns an empty string.

5. Close the VI by selecting Close from the File menu. Save the VI in your MYWORK directory as Parse String.vi.

PREV NEXT

File Input/Output

File input and output (I/O) operations retrieve information from and store information in a disk file. LabVIEW has a number of very versatile file I/O functions, as well as some simple functions that take care of almost all aspects of file I/O in one shot. We'll talk about the simple file functions in this chapter. All are located in the Programming>>File I/O subpalette of the Functions palette.

How They Work

The File functions expect a file path input, which looks kind of like a string. A path is a specific data type that provides a platform-specific way to enter a path to a file. We talked about them briefly in Chapter 4, and they'll come up again in Chapter 14, "Advanced LabVIEW Data Concepts." If you don't wire a file path, the File functions will pop up a dialog box asking you to select or enter a filename. When called, the File functions open or create a file, read or write the data, and then close the file. The files created with the VIs we'll talk about now are just ordinary text files. Once you have written data to a file, you can open the file using any word processing program to see your data.

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

Express Writing and Reading of Measurement Files

For quick and interactive configuration of data file I/O operations in LabVIEW, consider using the Write To Measurement File and Read From Measurement File Express VIs, shown in Figure 9.29 and Figure 9.30.

Figure 9.29. Write To Measurement File

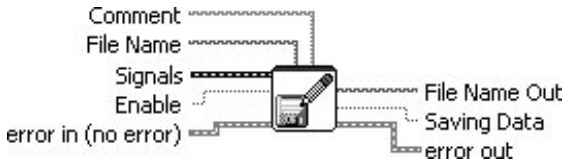
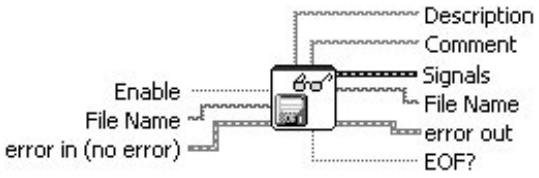


Figure 9.30. Read From Measurement File



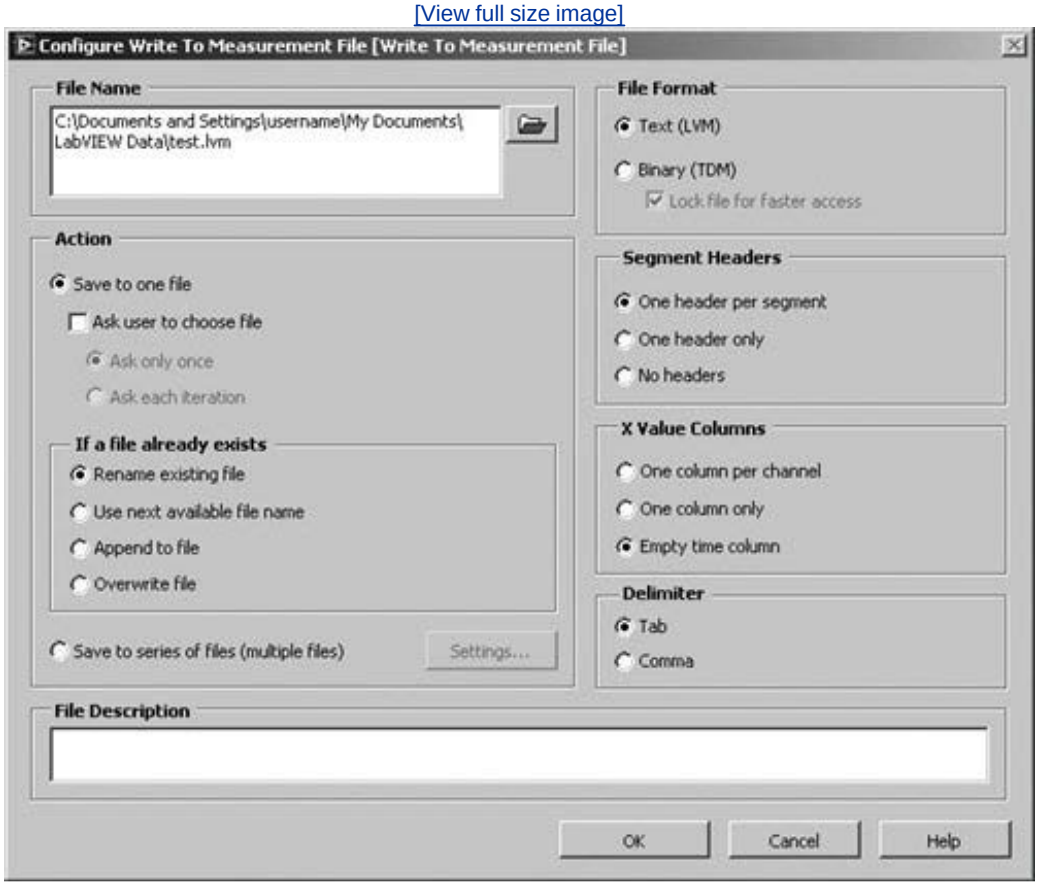
These VI pass data as the dynamic data type (which we learned about in Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs"), so they can handle just about any type of file containing measurement data.

To configure these Express VIs after they are placed on the block diagram, open the configuration dialog by double-clicking on the subVI or selecting Properties from the pop-up menu.

From the Write To Measurement File Express VI configuration dialog (shown in Figure 9.31), you can specify how a file will be formatted, and what data to store in the file.

Figure 9.31. Write To Measurement File configuration dialog

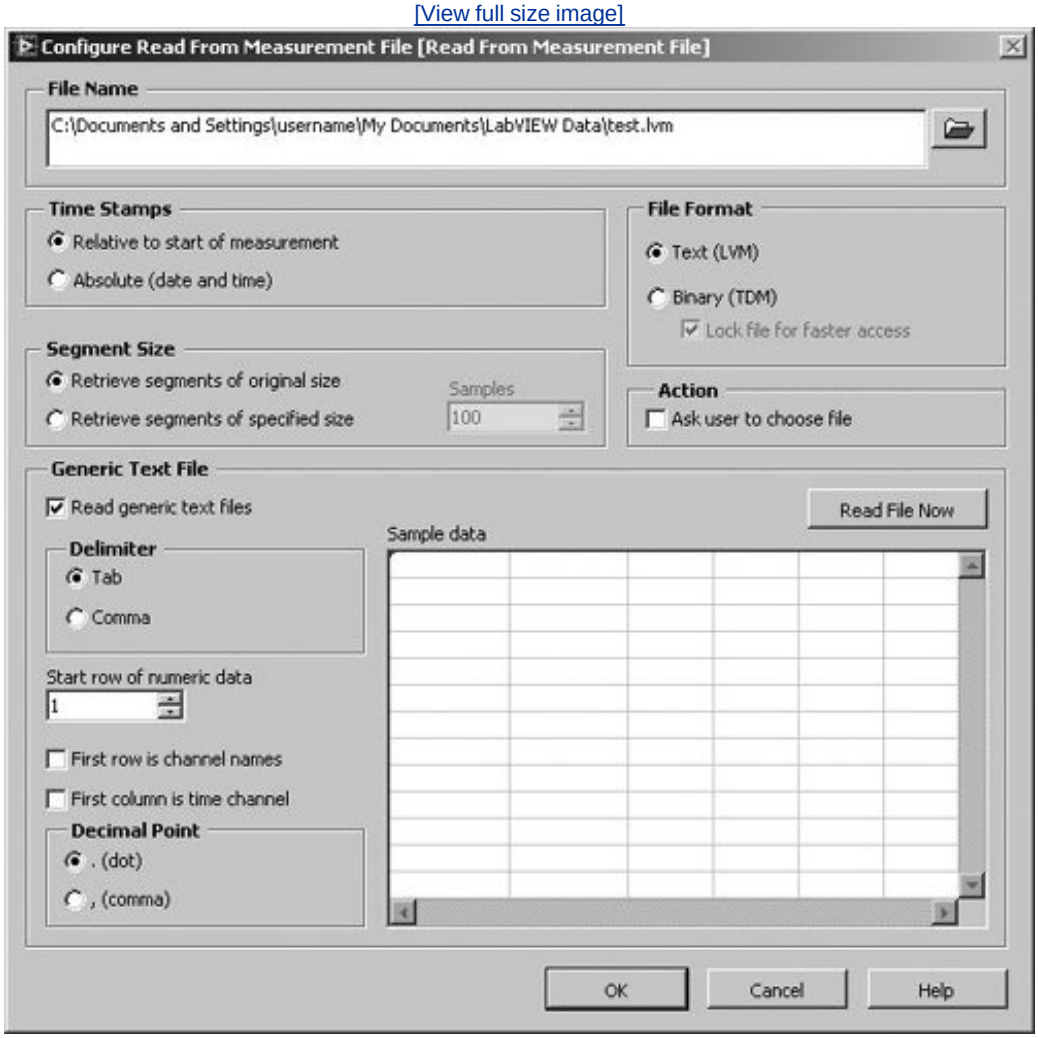
[View full size image]



From the Read From Measurement File Express VI configuration dialog (shown in Figure 9.32), you can specify the format of the file to be read. If you press the Read File Now button, LabVIEW will read the file once and populate the Sample Data table to show you whether the read operation worked successfully.

Figure 9.32. Read From Measurement File configuration dialog

[View full size image]



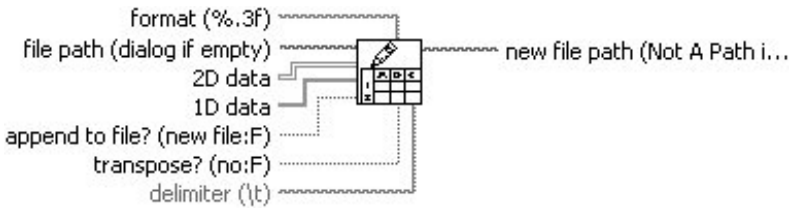
Writing and Reading Spreadsheet Files

If the Express VIs do not provide you with enough flexibility, you can read and write directly to

spreadsheet text files with Write To Spreadsheet File and Read From Spreadsheet File.

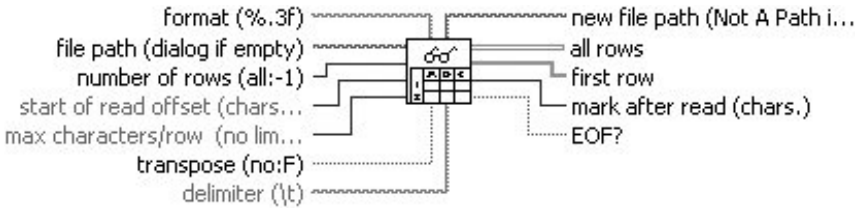
Write To Spreadsheet File converts a 2D or 1D array of single-precision numbers to a text string, and then writes the string to a new byte stream file or appends the string to an existing file. You can optionally transpose the data. Do not wire inputs for both 1D and 2D data (or one will be ignored). The text files created by this VI are readable by most spreadsheet applications.

Figure 9.33. Write To Spreadsheet File.vi



Read From Spreadsheet File reads a specified number of lines or rows from a numeric text file, beginning at a specified character offset, and converts the data to a 2D single-precision array of numbers (see Figure 9.34). You can optionally transpose the array. This VI will read spreadsheet files saved in text format.

Figure 9.34. Read From Spreadsheet File.vi.

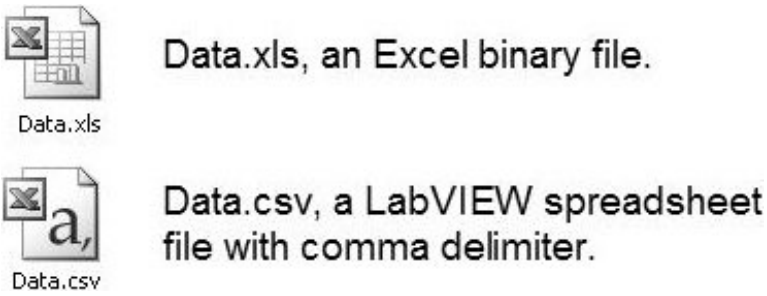


Write To Spreadsheet File.vi and Read From Spreadsheet File.vi do not have `Error In` *or* `Error Out` *terminals. They do all of their error handling internally, and will open an error dialog if an error does occur inside the VI. This behavior can be nice for simple applications, but is generally undesirable for advanced applications.*

Spreadsheet Files: CSV Versus XLSWhat's the Difference?

Write To Spreadsheet File.vi and Read From Spreadsheet File.vi write and read data as text files using a delimiter (usually a comma or tab) to separate columns and an end of line character (\r, \n, or \r\n) to separate rows. Commonly, a spreadsheet file that uses commas as the delimiter is referred to as a Comma Separated Values (CSV) file and is routinely saved with the filename extension ".csv," which Microsoft Excel, OpenOffice.org Calc, and other spreadsheet applications will recognize. For example, on a computer with Microsoft Excel installed, you can double-click on a CSV file and it will open in Excel. In fact, CSV files (if named with a ".csv" file extension) will have an icon, very similar to Excel (".xls") files, as shown in Figure 9.35.

Figure 9.35. Excel icons for XLS and CSV files



However, it is important to realize that if you save a file in Excel (or another spreadsheet program), it will not save the file as a CSV (unless you specify that option); but rather, it will be saved as a binary file (usually with a different file extension, such as ".xls" for Excel files). The resulting binary file *cannot be read* using Read From Spreadsheet File.vi.

The bottom line is that users of your application may say that they want data stored in Excel files, but what they really mean is that *they want data stored in a file that Excel can read*. Generate CSV files for your users, and they will probably be very happy with the resultand sometimes they won't even know the difference.

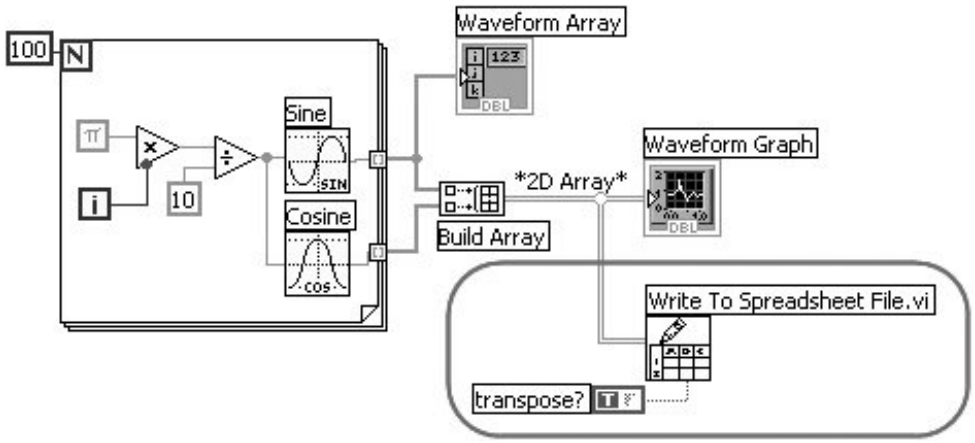
Activity 9-3: Writing to a Spreadsheet File

You will modify an existing VI to save data to a new file in ASCII format. Later you can access this file from a spreadsheet application.

1. Open Graph Sine Array.vi, which you built in Chapter 8. If you didn't finish building that VI, you can find a finished version in EVERYONE\CH08. As you recall, this VI generates two data arrays and plots them on a graph. You will modify this VI to write the two arrays to a file in which each column contains a data array.

2. Open the diagram of Graph Sine Array.vi and modify the VI by adding the diagram code shown inside the oval, as shown in Figure 9.36.

Figure 9.36. Graph Sine Array.vi block diagram, with modifications you will perform during this activity



Write to Spreadsheet File VI

The Write To Spreadsheet File VI (Programming>>File I/O palette) converts the 2D array to a spreadsheet string and writes it to a file. If no path name is specified (as in this activity), then a file dialog box will pop up and prompt you for a file name.

Boolean Constant

The Boolean Constant (Programming>>Boolean palette) controls whether or not the 2D array is transposed before it is written to file. To change it to TRUE, click on the constant with the Operating tool. In this case, you do want the data transposed because the data arrays are row specific (each row of the 2D array is a data array). Because you want each column of the spreadsheet file to contain data for one waveform, the 2D array must first be transposed.

3. Return to the front panel and run the VI. After the data arrays have been generated, a file dialog box will prompt you for the file name of the new file you are creating. Type in a file name (or if you don't see this option, click on the "New . . ." button from the dialog box, and choose "File") and click the OK button. Remember the name and location of the file, as you will read in the data in the next exercise.

Do not attempt to write data files in VI libraries with the file I/0 VIs. Doing so may overwrite your library and destroy your previous work.

4. Save the VI in your `MYWORK` directory, name it Graph Sine Array to File.vi, and close the VI.

5. Use spreadsheet software if you have it, or a simple text editor, to open and view the file you just created. You should see two columns of 100 elements each.

Activity 9-4: Reading from the Spreadsheet File

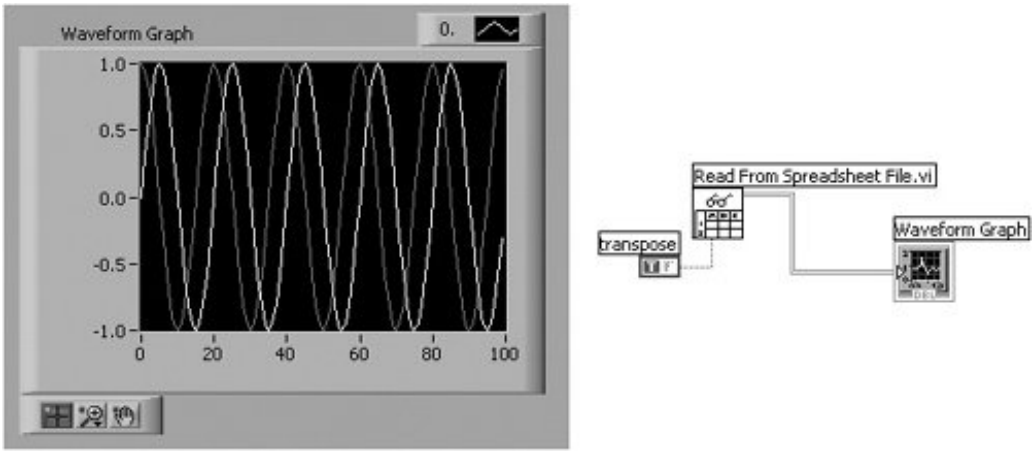
You will write a VI to read in the data from the file written in the last exercise and plot it on a graph.

1. Open a new VI and place a waveform graph on its front panel. Make sure autoscaling is on.

2. Create the little block diagram shown in Figure 9.37. Use the Read From Spreadsheet File function to bring in data and display it on the graph.

Figure 9.37. Front panel and block diagram of the VI you will create during this activity

[View full size image]



3. Using the TRUE Boolean Constant, you must transpose the array when you read it in, because graphs plot data by row and it has been stored in the file by column. Note that if you hadn't transposed the data in the last exercise to store it in columns in the file, you wouldn't have to transpose it back now.

4. Run the VI. Because you are not providing a file path, a dialog box will prompt you to enter a filename. Select the file you created in Activity 9-3.

   The VI will read the data from the file and plot both waveforms on the graph.

5. Save the VI in your `MYWORK` directory as Read File.vi.

More Writing and Reading of Files

You've seen how to read and write spreadsheet files or measurement data (using the dynamic data type). These functions are very easy to use.

Now we will discuss basic file I/O functions for working with text and binary files.

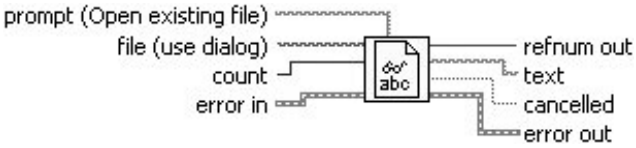
Writing and Reading Text Files

If you need to read or write a plain text file, you can use the LabVIEW functions on the Programming>>File I/O palette Write To Text File and Read From Text File.

Use the Write To Text File (shown in Figure 9.38) to write a string of characters to a file. Use Read From Text File (shown in Figure 9.39) to read a string of characters from a file.

Figure 9.38. Write To Text File

Figure 9.39. Read From Text File



The file input of Write To Text File and Read From Text File are path data types. However, you can wire a file refnum (which is discussed in Chapter 14, "Advanced LabVIEW Data Concepts") instead and the file input will adapt to a refnum type. These VIs are smart, in the sense that they can accept either a path or a refnum as their input.

It is important to note, that when you wire a path input into these VIs, if you do not connect a wire to the refnum out terminal, then you do not need to close the file refnum using the Close File function (which is discussed in Chapter 14). Write To Text File and Read From Text File are smart and will automatically close the file, if you do not wire anything to the refnum out terminal. The same is true for Write To Binary File and Read From Binary File

When you call Write To Text File and Read From Text File repeatedly (in a While Loop, for example), they will "remember" where they left off. For the Write To Text File function, this means that each time you write a string of characters, they are appended to the last string of characters that you wrote. For the Read From Text File function, this means that when you read some number of characters (specified by the count input), it will read that number of characters starting where you left off, last time you read the file. LabVIEW keeps track of where you left off automatically, storing this position in the *file marker*, to make your life easier. As we will learn next, we can explicitly set and get the position of the *file marker*, which makes it possible to have random access to data stored in large files.

Writing and Reading Binary Files

Just as you can read and write plain text files, LabVIEW also gives you functions to read and write binary files.

Binary files are efficient for storing large amounts of non-text data because, generally, they take up much less disk space than text files. For example, storing the number "3.14159265358979" in a binary file might only take up eight bytes (as a double-precision number), whereas in a text file it would take up 16 bytes (one byte for each character).

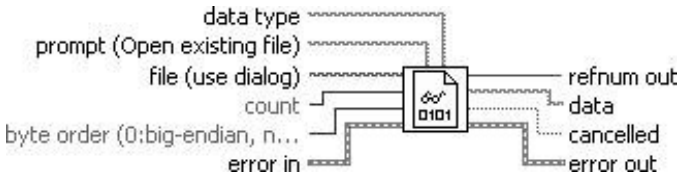
On the other hand, you can't easily open binary files with another application (like Word or Excel) because those other applications don't "know" how the data was formatted. Because a binary file could contain any type of data (for example, an array of integers, or a cluster of strings), it's up to you to specify how to read back that data. When you write a binary file with LabVIEW, you'll generally use LabVIEW to read back that binary file.

The Write To Binary File function is very easy to use and works just like the Write To Text File function we saw earlier, except that you can wire any kind of data you like to the data input (see Figure 9.40). The file input can be either a path or a file refnum.

Figure 9.40. Write To Binary File

The Read From Binary File is also easy to use, and is similar to the Read From Text File function, except that you *must* tell Read From Binary File what kind of binary data is stored in the file you are trying to read (see Figure 9.41). You do this by wiring the right kind of data to the data type input. You must wire the same data type that you used to write the file. Otherwise, you'll get garbage results in the data output.

Figure 9.41. Read From Binary File



For example, if you wrote arrays of Booleans to your binary data file, then you need to wire an array of Booleans to the data type input.

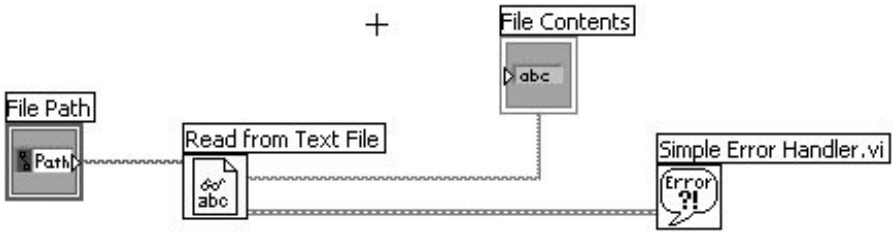
Activity 9-5: Reading a Text File

You will create a VI that reads the entire contents of a text file and displays the contents in a string indicator and the file size in a numeric indicator.

1. Build the front panel shown in Figure 9.42.

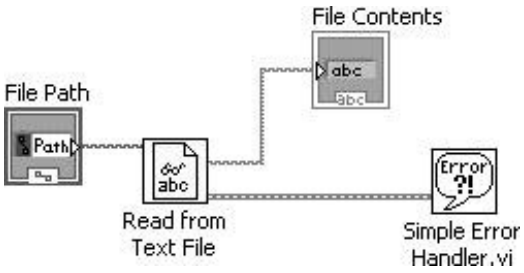
   The VI will read the contents of the file specified by the File Path control and display the output in the string indicator. The digital indicator will display the file's length.

Figure 9.42. Front panel of the VI you will create during this activity



2. Build the block diagram in Figure 9.43.

Figure 9.43. Block diagram of the VI you will create during this activity



Read from Text File Function

Read From Text File function (Programming>>File I/O palette) returns the contents of a text file in a string.

Simple Error Handler.vi

Simple Error Handler.vi (Programming>>Dialog & User Interface palette) displays a dialog if there is an error in one of the File I/O functions.

3. Return to the front panel and enter a path to a text file into the file path control. Make sure that you don't select a file that is too large (which will cause LabVIEW to use a lot of memory), or is a binary file (which will just display funny characters).

4. Run the VI and look at the file contents that are displayed in the string indicator, and the file size that is displayed in the numeric indicator.

5. Save and close the VI. Name it Read Text File.vi and place it in your `MYWORK` directory. Great jobyou're just made a text file viewer!

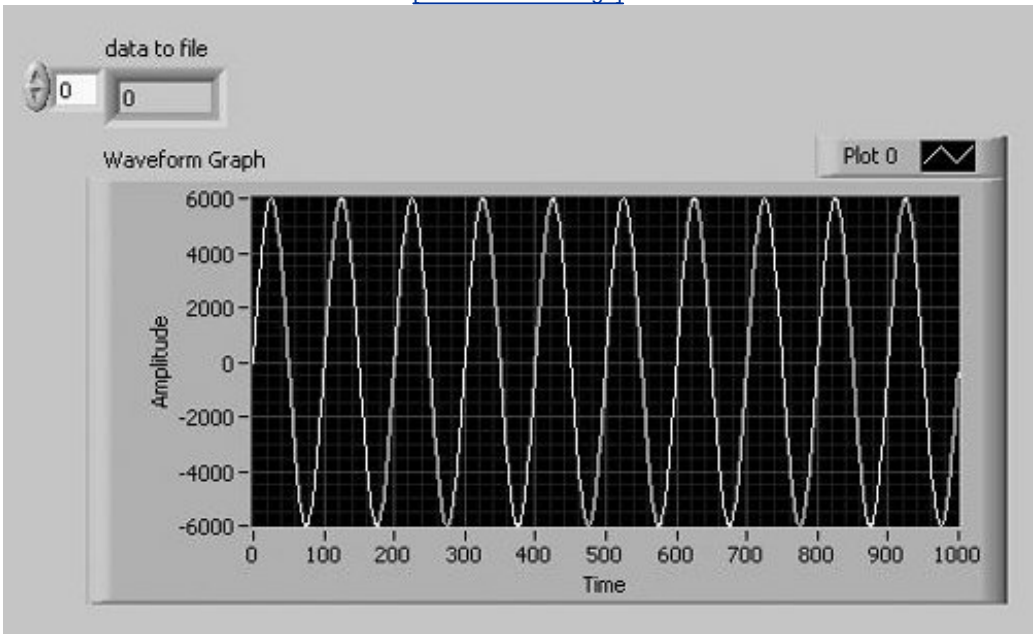
Activity 9-6: Writing and Reading Binary Files

As you saw in the previous activity, reading a text file is pretty trivial. In this activity, you'll build a VI that writes to a binary file and then another VI that can read those binary files.

1. First, let's create the VI that writes a binary file. Build a front panel like the one shown in Figure 9.44, with one chart and one array of numbers.

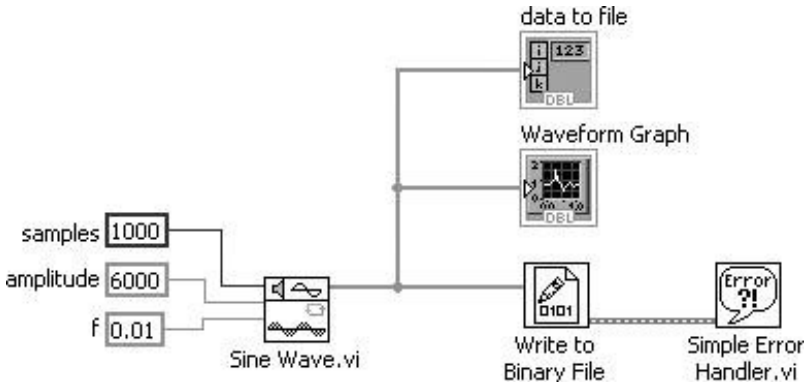
Figure 9.44. Write To Binary File front panel

[View full size image]



2. Build a block diagram like the one shown in Figure 9.45.

Figure 9.45. Write To Binary File block diagram

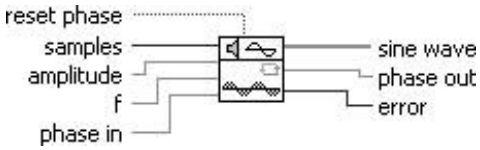


We'll make our binary data by generating an array of points that represents sinewave. To do this, we can use a function called Sine Wave.vi. You can find this function in the Programming>>Signal Processing>>Signal Generation palette, as shown in Figure 9.45.

3. Save your VI as Write To Binary File.vi. Run your VI to test it. You should see the data plotted, and you will be prompted to save to a file. You call your file "My Data.dat" or something similar.

   Sine Wave.vi (found on the Programming>>Signal Processing>>Signal Generation palette) generates an array containing a sinewave (see Figure 9.46).

Figure 9.46. Sine Wave.vi



4. Now let's build the VI that can read the binary file you just created. Create a front panel and block diagram like the ones shown in Figures 9.47 and 9.48.

Figure 9.47. Read From Binary File.vi front panel

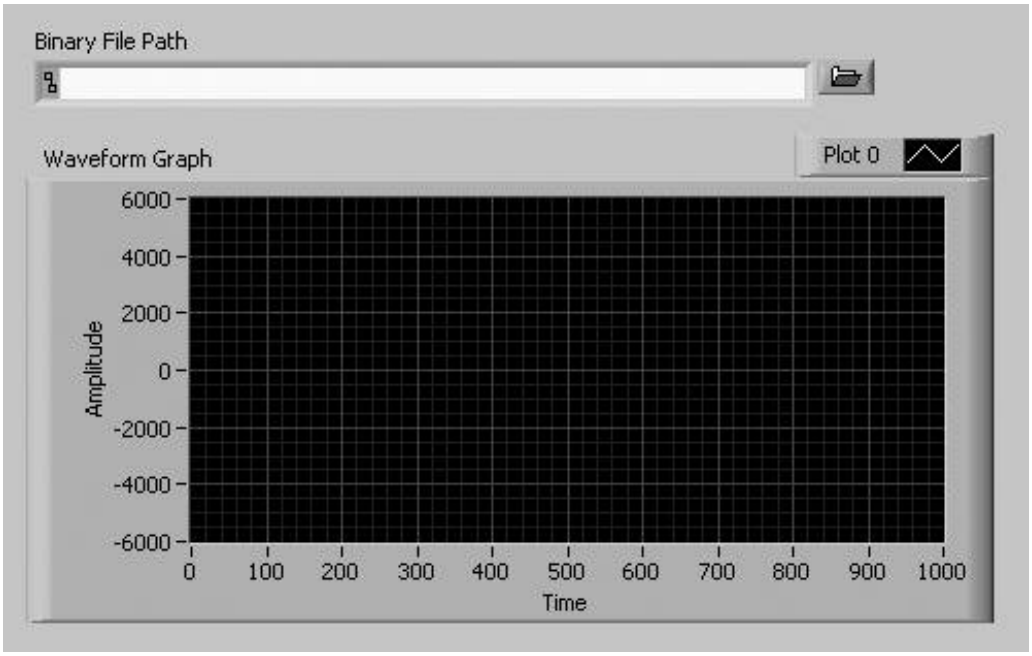
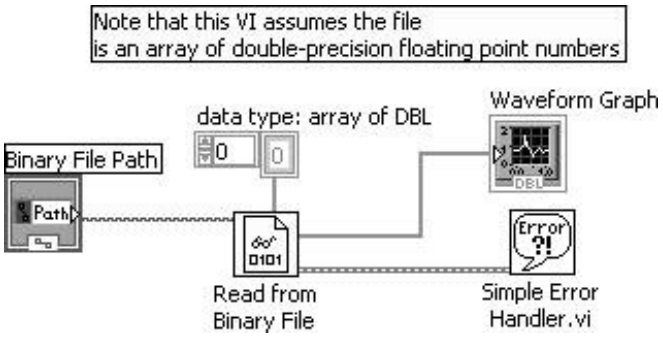


Figure 9.48. Read From Binary File.vi block diagram



5. For the block diagram, use the Read From Binary File function from the Programming>>File IO palette.

   Notice that you MUST wire the "data type" input. Unlike text files, when you read a binary file, you have to specify what kind of data is inside the file. Because we know that this file was written as an array of DBL, we create an array constant on the block diagram of DBL.

   To create the array constant "data type: array of DBL," you do the following:

   a. From the Programming>>Array palette, choose Array Constant and place the blank array constant on the block diagram.

   b. From the Programming>>Numeric palette, choose Numeric Constant and place this inside the array box.

c. You have an array now, but the default numeric datatype is an integer (I32). Right-click on the numeric constant inside the array and select Representation>>DBL.

d. Now wire this array constant to the "data type" input on the Read From Binary File.

6. Save your VI as Read From Binary File.vi and run it. When prompted, select the binary file you saved earlier.

7. As an interesting exercise, use the Read From Text File.vi you created and open the binary file.

PREV NEXT

Wrap It Up!

LabVIEW contains many functions for manipulating strings. These functions can be found in the String palette. With them, you can determine string length, combine two strings, peel off a string subset, convert a string to a number (or *vice versa*), and many other useful things.

"\" codes display mode allows you to view non-printable characters in strings. And, string controls and indicators have other display options and components to suit your application's needs.

Listboxes and tables allow you to display 2D Arrays of strings, as well as allow users to edit and select subsets of the arrays.

Regular expressions allow you to find anything inside of a string. The Match Pattern function and the more powerful Match Regular Expression function do all the heavy liftingyou just need to know how to create regular expressions in the special "regular expression syntax" to describe the pattern you wish to match.

Using the functions in the File I/O palette, you can write data to or read data from a disk file. Write To Spreadsheet File will save a numerical array as a text spreadsheet string in a file. Read From Spreadsheet File can then read that file back into LabVIEW. You can also read and write dynamic data types with the Write To Measurement File and Read From Measurement File Express VIs. For more advanced applications, you can read and write text files, binary files, and perform low-level file IO.

A special kind of file in software applications is the configuration file (sometimes called INI file). LabVIEW gives you a set of functions for creating, reading, and writing your own configuration files.

Congratulations! Go out and celebrate today! You've mastered the fundamentals of LabVIEW! You have a strong foundation now and should have the background to investigate almost any LabVIEW topic that interests you. The "Advanced" section of this book, coming up next, will teach you about many of the very cool, more complex features LabVIEW contains that make your programming job easier, so stay tuned for more exciting LabVIEW adventures!

Additional Activities

Activity 9-7: Temperatures and Time Stamps

Build a VI that takes 50 temperature readings inside a loop, once every 0.25 seconds, and plots each on a chart. It also converts each reading to a string, and then concatenates that string with a Tab character, a time stamp, and an end of line character. The VI writes all of this data to a file. Save the VI as Temperature Log.vi.

- *Use the Tab and End of Line constants in the String palette.*
- *Use Concatenate Strings to put all of the strings together.*
- *Use Write Characters To File to save the data.*
- *You can write data to a file one line at a time, but it is much faster and more efficient to collect all of the data in one big string using shift registers and Concatenate Strings, and then write it all to a file at one time.*

You can look at your file using any word processing program, but it should look something like this:

```
78.9  11:34:38
79.0  11:34:39
79.0  11:34:50
```

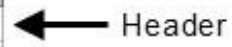
Activity 9-8: Spreadsheet Exercise

Build a VI that generates a 2D array (three rows x 100 columns) of random numbers and writes the transposed data to a spreadsheet file (see Figure 9.49). (Save it with a ".csv" file extension so that it can be opened in your spreadsheet application, by double-clicking it.) The file should contain a header

for each column as shown. Use the VIs from the Programming>>String and Programming>>File I/O palettes for this activity. Save the VI as Spreadsheet Exercise.vi.

Figure 9.49. Data generated during this activity, displayed in a spreadsheet application

	A	B	C
1	Waveform 1	Waveform 2	Waveform 3
2	0.084	0.567	0.412
3	0.907	0.858	0.745
4	0.825	0.479	0.314
5	0.839	0.479	0.55
6	0.578	0.799	0.916
7	0.159	0.183	0.017
8	0.138	0.363	0.152
9	0.149	0.693	0.643
10	0.75	0.332	0.875
11	0.913	0.54	0.544



PREV NEXT

10. Signal Measurement and Generation: Data Acquisition

Overview

This chapter will give you a primer course on data acquisition basics, which we touched on in Chapter 2, "Virtual Instrumentation: Hooking Your Computer Up to the Real World." Data acquisition is one of the main reasons why people use LabVIEWturning their computers into virtual instruments by gathering data from the real world. We'll take a look at the various options you have for taking or making data using DAQ devices. You'll also learn some signal theory and about the kind of hardware used for these systems.

Goals

- Finally find the meaning of all those data acquisition acronyms that everyone thinks you know
- Become familiar with the hardware options you have for acquiring or sending data
- Learn some signal theory, including the classification of signals, measurement types, signal conditioning, and sampling
- Get some hints on picking and installing a DAQ device that suits your needs
- Learn how to set up simulated devices so you can still write DAQ programs without hardware

Key Terms

- DAQ
- Signals
- Analog
- Digital
- Frequency
- Grounded signal
- Floating signal
- Ground reference
- Sampling rate
- Nyquist frequency
- Signal conditioning
- MAX
- NI-DAQmx
- Differential measurement
- Single-ended measurement
- Simulated device
- Scale
- Task

DAQ and Other Data Acquisition Acronyms

"Let's go ahead and apply CASE tools and UML to designing the PCI interface using that new XML standard."

Admit it, how many times has someone mentioned an acronym in a technical discussion and everyone pretends to understand because nobody wants to ask what it stands for and look ignorant? Well, here's your chance to see what all the acronyms in this chapter stand for and where they came from. Use this list to put your colleagues to the test!

AC: Alternating Current. This acronym originally referred to how a device was powered, with AC being the plug in the wall and DC (direct current) being batteries. Now it's used more generally to refer to any kind of signal (not just current) that varies "rapidly" (whatever you want that to mean) with time.

AC/DC: A rockin' Aussie band. (Rock historians will note that the band was founded by two Scottish brothers, Angus and Malcom Young, who immigrated to Australia ten years prior to forming the band.)

ADC: or A/D Analog-to-Digital Conversion. This conversion takes a real-world analog signal, and converts it to a digital form (as a series of bits) that the computer can understand. Many times the chip used to perform this operation is called "the ADC."

DAQ: Data AcQuisition. This little phrase just refers to collecting data in general, usually by performing an A/D conversion. Its meaning is sometimes expanded to include, as in this book, data generation. Don't confuse DAQ and DAC, which sound the same when pronounced in English. (DAC, or D/A, stands for Digital-to-Analog Conversion, usually referring to the chip that does this.)

DC: Direct Current. The opposite of AC. No longer refers to current specifically. Sometimes people use DC to mean a constant signal of zero frequency. In other cases, such as in DAQ terminology, DC also refers to a very low-frequency signal, such as something that varies less than once a second.

DMA: Direct Memory Access. You can use plug-in DAQ devices that have built-in DMA, or buy a separate DMA board. DMA lets you throw the data you're acquiring directly into the computer's RAM (there we go, another acronym), thus increasing data transfer speed. Without DMA, you still acquire data into memory, but it takes more steps and more time because the software has to direct it there.

IEEE: Institute of Electrical and Electronics Engineers. An international non-profit, professional association for the advancement of technology. IEEE sets standards in a variety of industries. For example, in this chapter, we will learn about TEDS, and in Chapter 14, "Advanced LabVIEW Data Concepts," we will learn about GPIB and Ethernetall of which are IEEE standards.

MAX: Measurement & Automation Explorer software from National Instruments. Your one-stop-shop for setting up and testing your National Instruments DAQ hardware and software, and your external instruments (on Windows platforms only).

MXI-3: Multisystem eXtension Interface (version 3) is a high-speed serial interface bus for bridging PCI buses. For example, it allows you to chain multiple PXI chassis together and allows a single computer to control the cards on all the chassis.

NI-DAQmx: NI-DAQmx is the latest DAQ driver from National Instruments. It supersedes Traditional NI-DAQ, the previous DAQ driver.

PXI: PCI eXtensions for Instrumentation. PCI ("Peripheral Component Interconnect") is a standard bus, used on most computers for plugging in dedicated device cards. PXI refers to an open hardware architecture, embraced by National Instruments, for integrating high-performance, modular components for data acquisition, instrument control, image processing, and more.

RTSI: Real-Time System Integration bus. The National Instruments timing bus that connects DAQ, Motion, Vision, and other devices directly for precise synchronization of functions.

SCC: Signal Conditioning Carriers. A compact, modular form factor for signal conditioning modules (for example, optically-isolated relays), sold by National Instruments.

SCXI: Signal Conditioning eXtensions for Instrumentation. A high-performance signal conditioning system devised by National Instruments, using an external chassis that contains I/O modules for signal conditioning, multiplexing, and so on. The chassis is wired into a DAQ device in the PC.

SISTA: Sometimes I'm Sick of These Acronyms. Just kidding.

TEDS: Transducer Electronic Data Sheets. An IEEE standard (Institute of Electrical and Electronic Engineers standard 488.2) for smart sensors that store their own calibration data.

TLA: Three-Letter Acronym. Not to be confused with FLA (Four-Letter Acronym), which is, ironically, a TLA!

USB: Universal Serial Bus, a standard bus on most PCs for connecting external peripherals.

VXI: Talk about acronym abuse; this is an acronym for an acronym: VME eXtensions for Instrumentation. VME stands for Versa-Modular Eurocard. VXI is a very high-performance system for instrumentation. You can buy VXI instruments that are small modules (instead of the regular big instrument with a front panel) that plug into a VXI chassis. The VXI chassis often has an embedded computer motherboard, so you don't have to use an external PC. VXI is an open industry standard, which means that companies besides National Instruments support it.

PREV NEXT

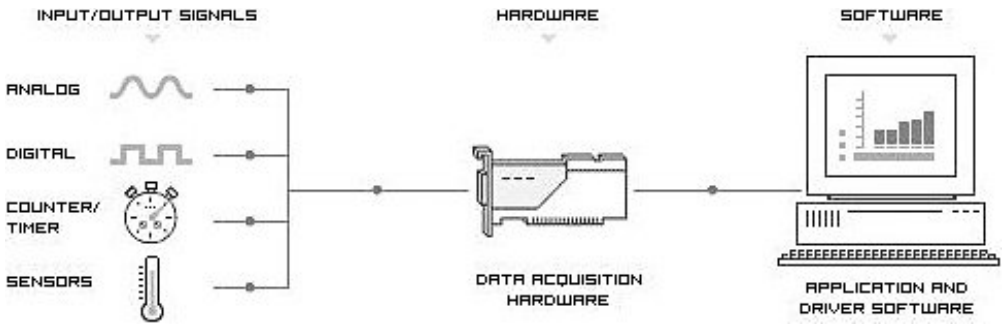
How to Connect Your Computer to the Real World

You've got a good PC, you're excited about using LabVIEW, and you're ready to build an application that will do something *outside* your computer Maybe you need to monitor the electroencephalogram (EEG, brain waves) of some research subjects, or plot force-versus-displacement curves to test the strength of some new plastic. Or perhaps you need something more elaborate, like a whole process control system for a semiconductor manufacturing facility, and you need to provide control signals to the plant.

Whatever your application, you need a way to gather data into your computer. Several solutions are usually possible, but the best solution will decidedly depend on what trade-offs you can afford. Before you run out and buy hardware, you need to analyze and understand what kind of signals you're trying to measure (in some cases, such as serial communication, you may not even need any additional hardware).

One of the first things you should decide when designing your data acquisition system is whether you're going to use traditional external instruments, such as a multimeter. Do you want a "physical" instrument (which we will discuss in Chapter 12, "Instrument Control in LabVIEW") to do some of the data acquisition and processing, or do you want to write a LabVIEW virtual instrument that will do everything via a plug-in DAQ device?

Figure 10.1. Connecting your computer to the real world using data acquisition hardware



Issues such as cost, scheduling, and flexibility will play a part in this decision. For example, if you wanted to read some low-voltage data, you might use a simple plug-in DAQ device. You could write a VI that is very specific to your application and creates the exact virtual instrument you need. And generally speaking, it's much cheaper to buy a plug-in DAQ device than a standalone instrument. On the other hand, if you already have an existing instrument you want to use (a multimeter, for example), it might be cheaper to use your existing meter to acquire the voltages and send the data to the computer via its communications port (which we will discuss in Chapter 12).

Finally, if you are planning to buy or already have bought a plug-in DAQ device, make good use of it! Most people don't fully realize the potential their computer has when a DAQ device is plugged into it. Need to view an AC signal? Wait, don't go borrow that oscilloscope; just look at your signal right on your screen with one of the DAQ example VIs that comes with LabVIEW! With one DAQ device, you can create as many virtual instruments as you need. And, when it's time to upgrade your plug-in device or move to another platform, you may not even need to change a thing in your block diagram. That's rightLabVIEW's DAQ VIs work (with a few exceptions) *independently of whatever device you have in your computer.*

Traditional, stand-alone instruments are very important to data acquisition systems. And, we have dedicated all of Chapter 12 to a discussion of how to connect your computer to your instruments and how to communicate with them from LabVIEW. This chapter will focus on basic signal theory and then on the selection and configuration of modular data acquisition hardware. Then, in Chapter 11, "Data Acquisition in LABVIEW," we will discuss how to use the LabVIEW DAQ VIs. And finally, we will segue into our instrumentation discussion in Chapter 12.

Now, let's learn about signals!

PREV NEXT

Signals 101

Before we delve completely into data acquisition, we want to talk a little about *what* you'll be acquiring. A *signal* is simply any physical quantity whose magnitude and variation with time (or occasionally some other variable) contain information.

Timing Is Everything

Although it may not be obvious at first, *time* is usually the most critical aspect of almost any measurement. Whether we want to observe how an engine's temperature changes over time, see what a filtered audio signal looks like, or close some valves when a gas mixture reaches its optimum ratio, time is the deciding factor in data acquisition and control. We want to know not just *what* happens, but *when.* Even so-called "DC" signals are not really steady-state; if they never changed over time, we would always know their constant value, so why would we want to measure them?

Timing is important in designing your data acquisition software for a couple of reasons. First, you need to figure out how to set the *sampling rate*, or how often your computer takes a measurement. Second, you need to be able to allocate processor time to other tasks such as file I/O.

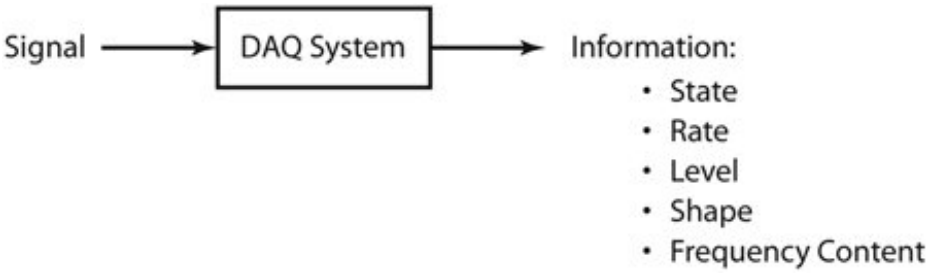
If you only need to read data once or twice a second or less and you don't need much timing accuracy between the sample points, you can probably use LabVIEW's timing functions to control the sampling rate "directly" from your program, by putting a Wait function in a VI that acquires one pointfor example, in a loop. For more precise applications or AC signals, you'll let the hardware and low-level software set the sampling rate by configuring your measurement on the DAQ device accordingly. We'll discuss these details along with some examples in Chapter 11.

Signal Classification

Let's say you want to take a measurement. For signal conditioning hardware to condition a signal, or for the DAQ device to measure it directly, you must first convert it to an electrical signal such as voltage or current. A *transducer* performs this conversion. For example, if you wish to measure temperature, you must somehow represent temperature as a voltage that the DAQ device can read. A variety of temperature transducers exist that use some physical properties of heat and materials to convert the temperature to an electrical signal.

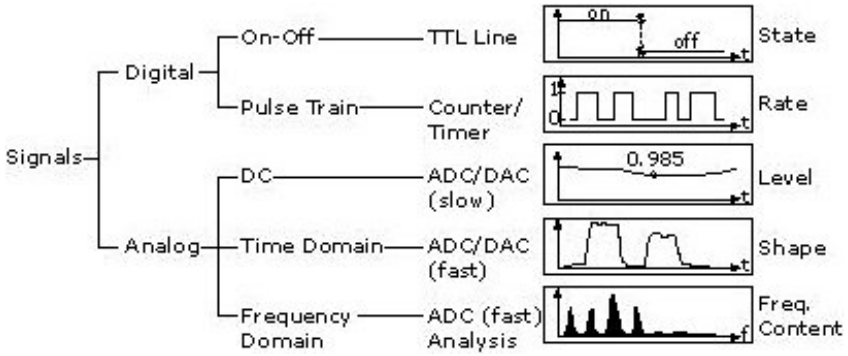
Once the physical quantity is in an electrical signal form, you can then measure the signals to extract some type of useful information conveyed through one or more of the following parameters: state, rate, level, shape, and frequency content (see Figure 10.2).

Figure 10.2. A DAQ System converts signals into information that your software can use



Strictly speaking, all signals are analog time-varying signals. However, to discuss signal measurement methods, you should classify a given signal as one of five signal types. Classify the signal by the way it conveys the needed information. First, you can classify any signal as *analog* or *digital*. A digital, or binary, signal has only two possible discrete levelsa high (on) level or low (off) level. An analog signal, on the other hand, contains information in the continuous variation of the signal with respect to time. Figure 10.3 shows a hierarchy of signal classifications.

Figure 10.3. Signal classification hierarchy

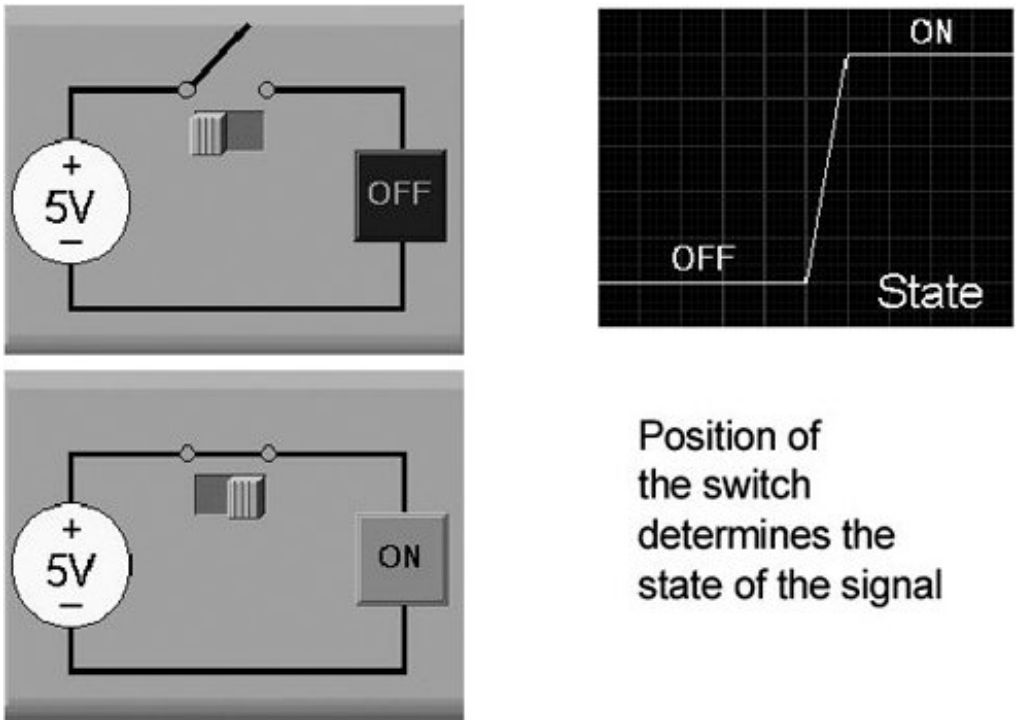


Engineers often classify digital signals into two more types and analog signals into three more types. The two digital signal types are the on-off signal and the *pulse train* signal. The three analog signal types are the *DC* signal, the *time domain* (or *AC*) signal, and the *frequency domain* signal. The two digital and three analog signal types are unique in the information each conveys. You will see that the five signal types closely parallel the five basic types of signal information: state, rate, level, shape, and frequency content.

Digital Signals

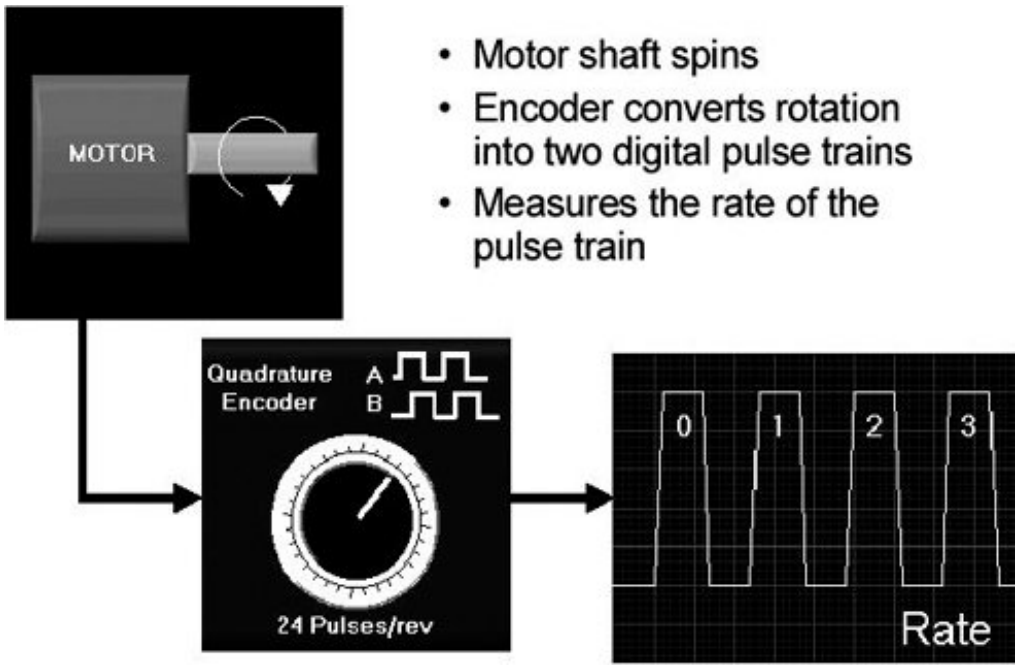
The first type of digital signal is the on-off, or state, signal. A state signal conveys information concerning the digital state of the signal. Therefore, the measurement hardware needed to measure this signal type is a simple digital state detector. The output of a transistor-transistor logic (TTL) switch is an example of a digital on-off signal. Another example is the state of an LED, as shown in Figure 10.4.

Figure 10.4. State example



The second type of digital signal is the pulse train, or rate, signal. This signal consists of a series of state transitions. Information is contained in the number of state transitions occurring, the rate at which the transitions occur, or the time between one or more state transitions. The output signal of an optical encoder mounted on the shaft of a motor is an example of a digital pulse train signal, as shown in Figure 10.5. In some instances, devices require a digital input for operation. For example, a stepper motor requires a series of digital pulses as an input to control the motor position and speed.

Figure 10.5. Rate example

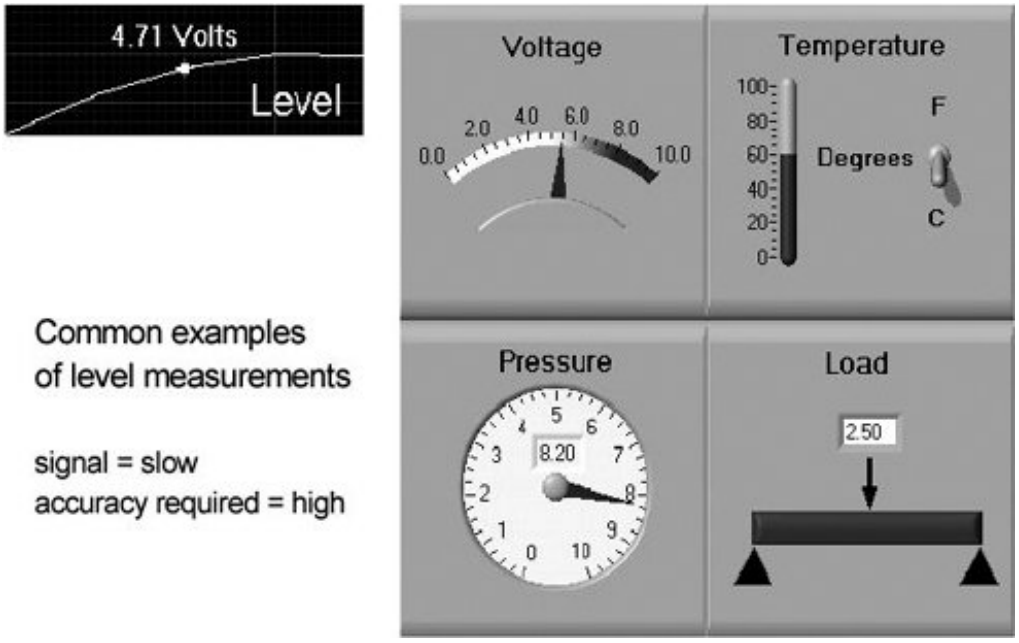


Analog Level Signals

Analog DC, or level, signals are static or slowly varying analog signals. The most important characteristic of the level signal is that the level, or amplitude, of the signal at a given instant conveys information of interest. Because the analog DC signal varies slowly, the accuracy of the measured level is of more concern than the time or rate at which you take the measurement. The plug-in DAQ device that measures DC signals operates as an analog-to-digital converter (ADC), which converts the analog electrical signal into a digital value for the computer to interpret.

As shown in Figure 10.6, common examples of DC signals include temperature, battery voltage, pressure, and static loads. In each case, the DAQ system monitors the signal and returns a single value indicating the magnitude of the signal at that time. Therefore, these signals are often displayed through LabVIEW indicators such as meters, gauges, strip charts, and numerical readouts.

Figure 10.6. Level examples



Your DAQ system should meet the following specifications when acquiring analog DC signals:

- High accuracy/resolution Accurately measure the signal level.
- Low bandwidth Sample the signal at low rates (software timing should be sufficient).

Analog Time Domain Signals

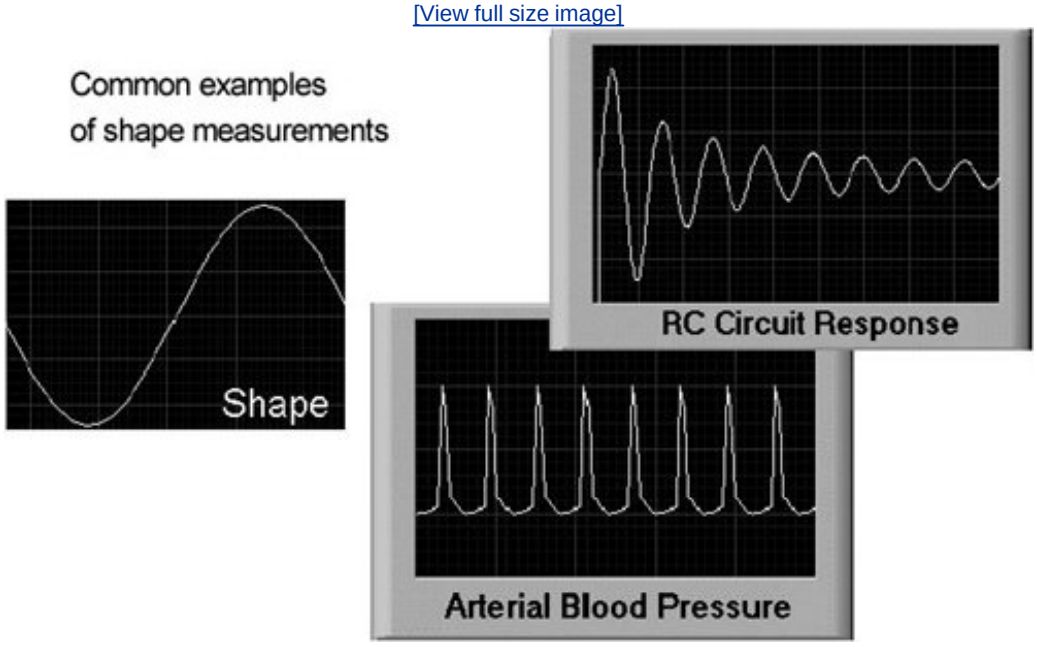
Analog time domain, or shape, signals differ from other signals in that they convey useful information not only in the signal level, but also in how this level varies with time. When measuring a shape signal, often referred to as a waveform, you are interested in some characteristics of the waveform shape, such as slope, locations and shapes of peaks, and so on.

To measure the shape of a time domain signal, you must take a precisely timed sequence of individual amplitude measurements, or points. These measurements must be taken at a rate that will adequately reproduce the shape of the waveform. Also, the series of measurements should start at the proper time, to guarantee that the useful part of the signal is acquired. Therefore, the plug-in DAQ device that measures time domain signals consists of an ADC, a sample clock, and a trigger. A sample clock accurately times the occurrence of each analog-to-digital (A/D) conversion. To ensure that the desired portion of the signal is acquired, the trigger starts the measurement at the proper time according to some external condition.

There are an unlimited number of different time domain signals, a few of which are shown in Figure 10.7. What they all have in common is that the shape of the waveform (level versus time) is the main feature of interest.

Figure 10.7. Shape examples

[View full size image]



Your DAQ system should meet the following specifications when acquiring analog time domain signals:

- Higher bandwidths Sample the signal at high rates.
- Accurate sample clock Sample the signal at precise intervals (hardware timing needed).
- Triggering Start taking the measurements at a precise time.

Analog Frequency Domain Signals

Analog frequency domain signals are similar to time domain signals because they also convey information on how the signals vary with time. However, the information extracted from a frequency domain signal is based on the signal frequency content, as opposed to the shape or time-varying characteristics of the waveform.

Like the time domain signal, the DAQ device used to measure a frequency domain signal must include an ADC, a sample clock, and a trigger to accurately capture the waveform. You can perform this type of digital signal processing (DSP) using application software or special DSP hardware designed to analyze the signal quickly and efficiently.

Your DAQ system should meet the following specifications when acquiring analog frequency domain signals:

- Higher bandwidths Sample the signal at high rates. The maximum frequency you can detect

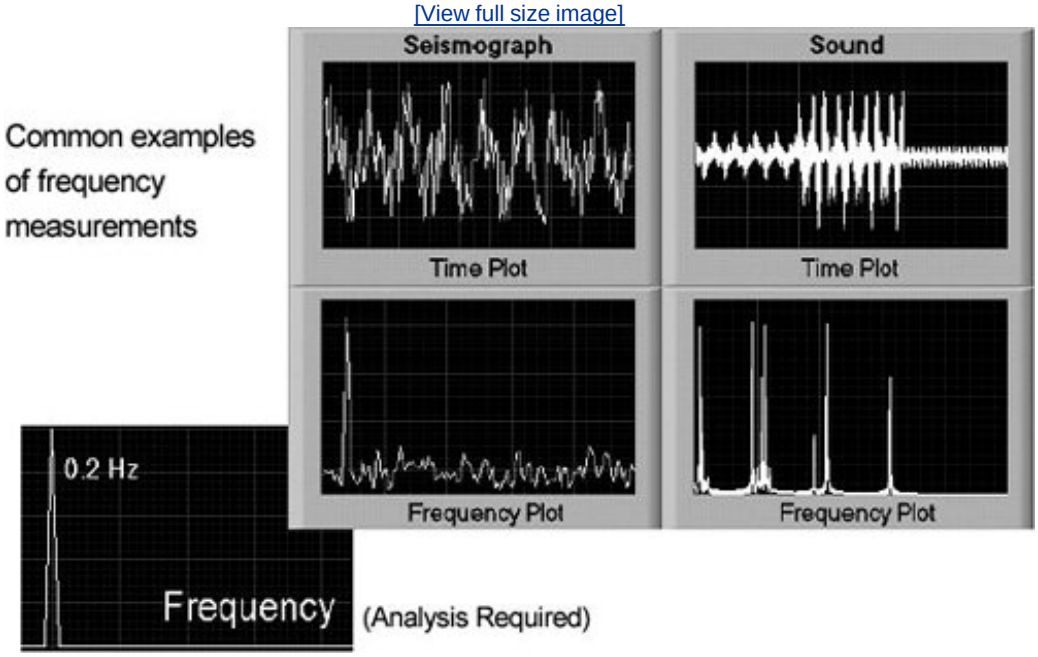
will always be less than half your sampling rate. This is part of Nyquist's Theorem, which we'll talk about later.

- Accurate sample clock Sample the signal at precise intervals (hardware timing needed).
- Triggering Start taking the measurements at a precise time.
- Analysis functions Convert time information to frequency information.

Figure 10.8 shows a few examples of frequency domain signals. Although you can analyze any signal in the frequency domain, certain signals and application areas lend themselves especially to this type of analysis. Among these areas are speech, acoustics, geophysical signals, vibration, and system transfer functions.

Figure 10.8. Frequency examples

[View full size image]



One SignalFive Measurement Perspectives

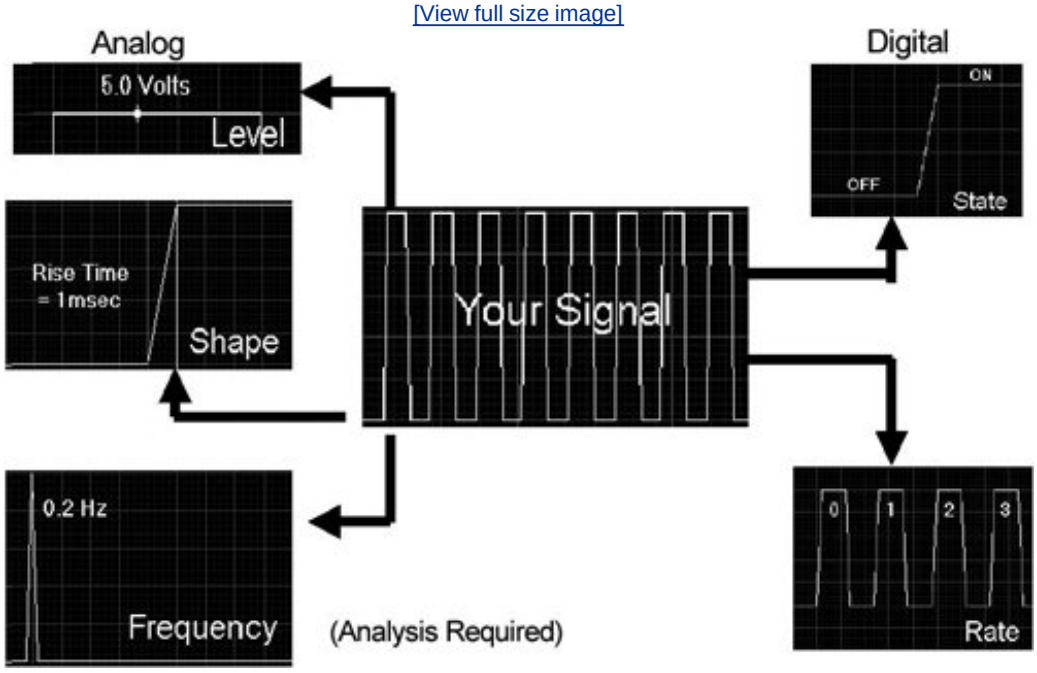
The five classifications of signals presented in this section are not mutually exclusive. A particular signal may convey more than one type of information. Therefore, a signal can be classified as more than one type of signal, and thus you can measure it in more than one way. You can use simpler measurement techniques with the digital on-off, pulse train, and DC signals because they are just simpler cases of the analog time domain signals.

You can measure the same signal with different types of systems, ranging from a simple digital input device to a sophisticated frequency analysis system. The measurement technique you choose depends on the information you want to extract from the signal. Look at Figure 10.9. It demonstrates

how one signala series of voltage pulsescan provide information for all five signal classes.

Figure 10.9. Five ways to measure the same signal

[View full size image]



Activity 10-1: Classifying Signals

Classify the following signals into one of the five signal types described earlierfrom the perspective of data acquisition. In some cases, a signal can have more than one classification. Choose the type that you think best matches the signal (the solutions are found at the end of this chapter, in the "Solutions to Activities" section). Circle the number on the left margin as follows:

Q1:

1. Analog DC
2. Analog AC
3. Digital on/off
4. Digital pulse/counter
5. Frequency

1 2 3 4 5	Voltage level of a battery
1 2 3 4 5	State of a solid-state relay
1 2 3 4 5	Data at your PC's parallel port during printing
1 2 3 4 5	Glitch or spike noise in power source
1 2 3 4 5	Transfer function of a digital filter
1 2 3 4 5	Data flowing over the Internet
1 2 3 4 5	Relative humidity outside
1 2 3 4 5	Car engine's RPM while driving around town
1 2 3 4 5	EEG (brain waves)
1 2 3 4 5	Speech through a microphone
1 2 3 4 5	Absolute pressure in an engine cylinder

Transducers

When you're setting up your DAQ system, remember that ultimately everything you're going to measure will have to become an electrical voltage or current. The way you convert measurable phenomena such as temperature, force, sound, light, humidity, etc., to an electrical signal is by using a *transducer*. Table 10.1 lists some common transducers used to convert physical phenomena into measurable quantities.

Table 10.1. Physical Phenomena and the Transducers Used to Measure Them

Phenomena	*Transducer*
Temperature	Thermocouples Resistance temperature detectors (RTDs) Thermistors Integrated circuit sensors
Light	Vacuum tube photosensors Photoconductive cells
Sound	Microphones
Force and pressure	Strain gauges Piezoelectric transducers Load cells
Position (displacement)	Potentiometers Linear voltage differential transformers (LVDT) Optical encoders
Fluid flow	Head meters Rotational flowmeters Ultrasonic flowmeters
pH	pH electrodes

Sensor Arrays: Imaging and Other Cool Stuff

A single sensor allows us to measure a physical quantity at a point in space. However, if we put lots of sensors, and sensors that measure different things, out in space, then we paint a much clearer picture (literally) of what is going on in the real world. This is called a *sensor array*.

One sensor array that we are all familiar with is the digital camera. Cameras are truly incredible sensing devices. They have an array of photo sensors that measure the intensity and wavelength of light emanating from a source, as a function of spatial position over a field of viewnamely, they take a picture! And, a picture is worth a thousand words.

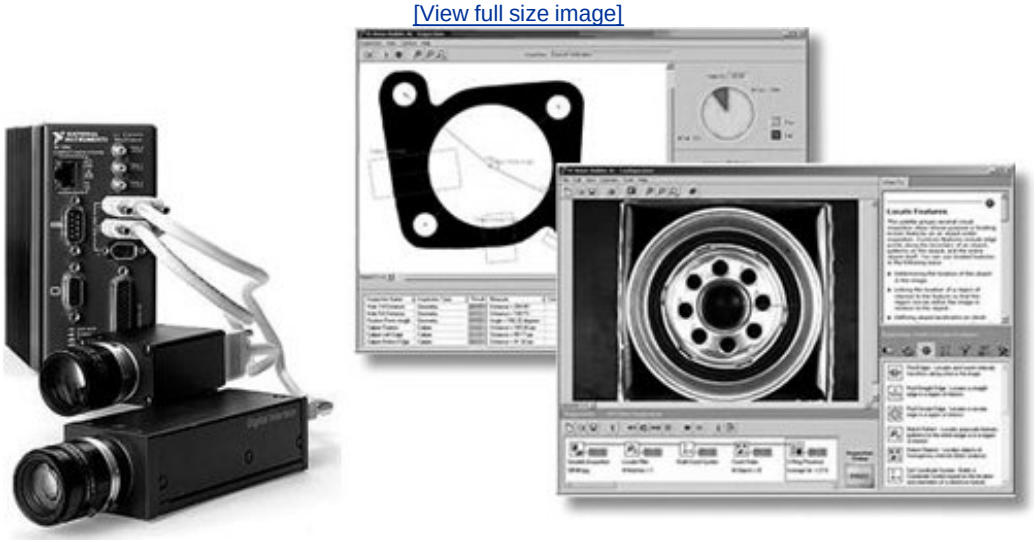
Black and white cameras acquire an optical intensity signal as a function of *X* position and *Y* position. Color cameras acquire an optical intensity signal as a function of *X* position and *Y* position and *spectral bin* (color).

Once we have image data, we can use image-processing algorithms to measure almost anything that the human eye and brain can detect, and more! You can take dimensional measurements, match patterns, read bar codes, read text, count objects, and a do variety of other useful things. Figure

10.10 shows some hardware and software used for vision and image processing.

Figure 10.10. Vision hardware and image processing

[View full size image]



Another useful sensor array is the microphone array, or *acoustic camera*. For example, Figure 10.11 shows a "picture" of the sound emanating from two SUVs, proving that those gas-guzzlers are noisy too!

Figure 10.11. Acoustic camera images (photo courtesy of Sound View Instruments AB)

You can make sensor arrays from any type of sensor, including temperature sensors, pressure sensors, strain gaugesthe sky's the limit!

Signal Conditioning

All right, so now that you've figured out what kind of signals you need to acquire, you can just plug the output of your transducers directly into the DAQ device, right? In perhaps 50% of the cases or more, NO! We may not always be aware of it, but we live in a very electrically noisy world60 Hz noise

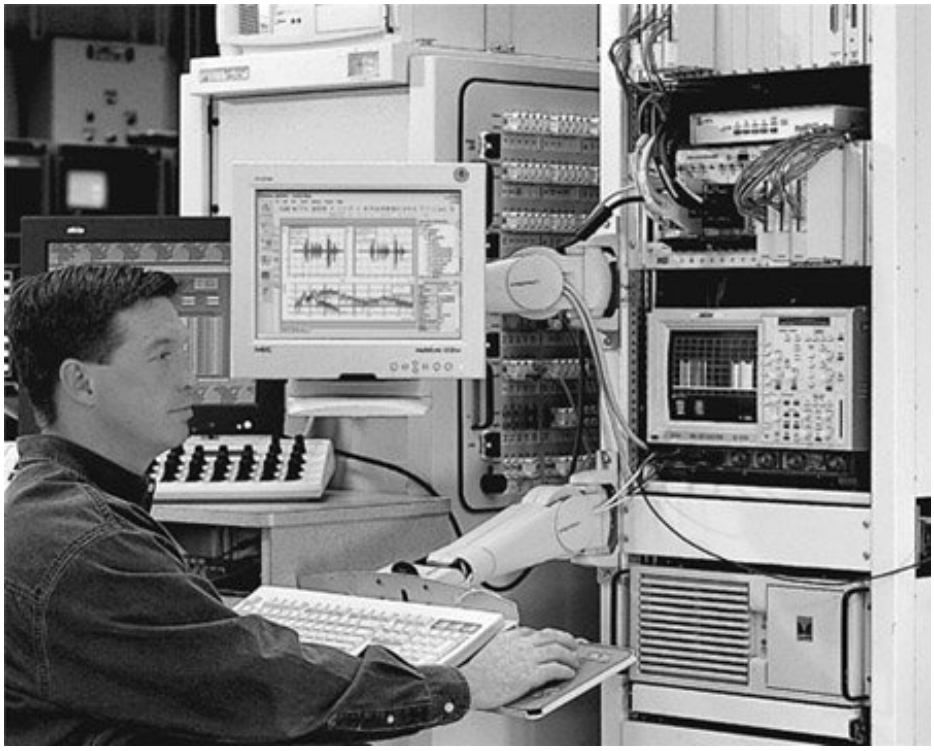
comes from AC power and lights; RF noise can come from fluorescent lights, cellphones, microwaves, electric motors, and so onit's everywhere. By the time your signal makes it to the DAQ device, it may have picked up so much noise or have so many other problems that it renders your measurement useless.

You usually need to perform some type of signal conditioning on analog signals that represent physical phenomena. What is signal conditioning, anyway? Simply put, it is a manipulation of your signal to prepare it for digitizing at the DAQ device. Your signal has to arrive as clean as possible, within the voltage (usually ±5 or 0 to 10 V) and current (usually 20 mA) limits of your DAQ device, with enough precision for your application. It's hard to be more specific unless you can specify what kind of transducers you're going to use. For example, signal conditioning for audio data from a microphone may involve nothing more than grounding the system properly and perhaps using a lowpass filter. On the other hand, if you want to measure ionization levels in a plasma chamber sitting at 800 V and you don't want to fry your computer, you'd need to provide some more complex circuitry that includes isolation amplifiers with a step-down gain.

For signals that need special conditioning, or for systems that have very many signals, National Instruments devised *SCX1* (Signal Conditioning eXtensions for Instrumentation). SCXI systems, such as the ones shown in Figures 10.12 and 10.13, provide a chassis where modular units can be inserted to build a custom system. These modular units include analog input multiplexers, analog output devices, "blank" breadboards, signal conditioning modules for thermocouples, and so on. For more information on SCXI, see the National Instruments catalog. Remember that you don't necessarily need SCXI to do signal conditioning.

Figure 10.12. An SCXI chassis with several signal conditioning modules can be connected to a PC with LabVIEW.

Figure 10.13. SCXI systems are often used in industrial rack-mount enclosures for high-channel count applications, as shown in this picture of "The Diadem Guy," hard at work. (photo courtesy of Daimler Chrysler Corporation)



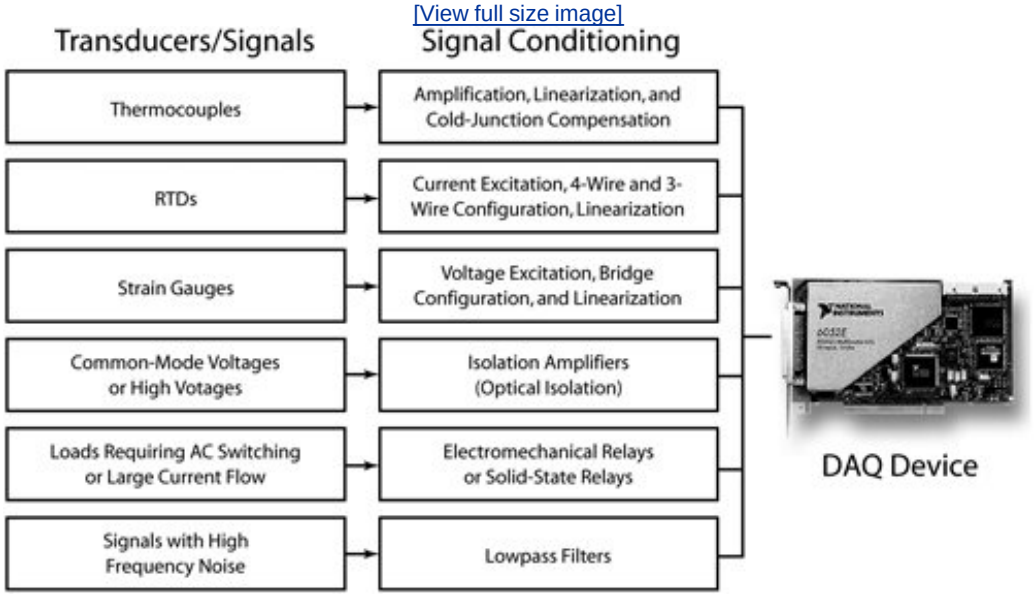
Some common types of signal conditioning are the following:

- Amplification
- Transducer excitation
- Linearization
- Isolation
- Filtering

Signal conditioning allows you to connect sensors and signals to your computer's DAQ device, as depicted in Figure 10.14.

Figure 10.14. Signal conditioning for different types of transducers and signals

[View full size image]



Finding a Common Ground

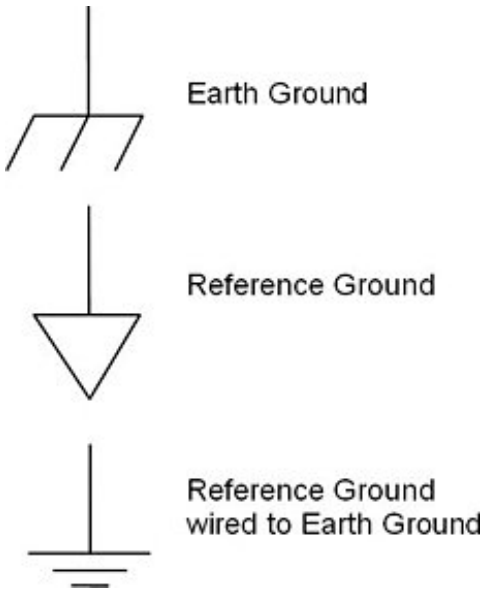
Some physical properties are absolute: luminosity or mass, for example (I know, I know, the physicists won't agreethings act weird when you approach the speed of light, and then almost nothing is absolute [some physicists even speculate that the speed of light is not constant]). But in any case, voltage is decidedly *not* absolute; it always requires a reference to be meaningful. Voltage is always the measure of a potential *difference* between two bodies. One of these bodies is usually picked to be the reference and is assigned "0 V." So to talk about a 3.47 V signal really means nothing unless we know with respect to what reference. If you've noticed, though, often a reference isn't specified. That's because the 0 V reference is usually the famous *ground*. Herein lays the source of much confusion, because "ground" is used in different contexts to refer to different reference potentials.

Earth ground refers to the potential of the earth below your feet. Most electrical outlets have a prong that connects to the earth ground, which is also usually wired into the building electrical system for safety. Many instruments also are "grounded" to this earth ground, so often you'll hear the term *system ground*. This is the ground that is usually tied to the third ground prong on electrical outlets. The main reason for this type of grounding is safety, and not because it is used as a reference potential. In fact, you can bet that no two sources that are connected to the earth ground are at the same reference level; the difference between them could easily be up to 10 volts. Conclusion: We're usually not talking about earth, or safety ground, when we need to specify a reference voltage.

Reference ground, sometimes called a return path or signal common, is usually the reference potential of interest. The common ground may or may not be wired to earth ground. The point is that many instruments, devices, and signal sources provide a reference (the negative terminal, common terminal, etc.) that gives meaning to the voltages we are measuring.

The ground symbols shown in Figure 10.15 are used in this book when you see wiring diagrams. Be aware, however, that you will find these same symbols used inconsistently among engineers.

Figure 10.15. Ground symbols



The DAQ devices in your computer are also expecting to measure voltage with respect to some reference. What reference should the DAQ device use? You have your choice, which will depend on the kind of signal source you're connecting. Signals can be classified into two broad categories, as follows:

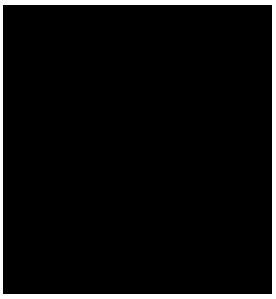
- Grounded
- Floating

Let's examine these categories a bit further:

Grounded Signal Source

A *grounded source* is one in which the voltage signals are referenced to a system ground, such as earth or building ground. Because they use the system ground, they share a common ground with the DAQ device. The most common examples of grounded sources are devices that plug into the building ground through wall outlets, such as signal generators and power supplies (see Figure 10.16).

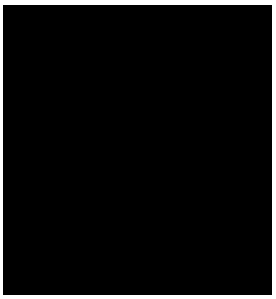
Figure 10.16. Grounded signal source



Floating Signal Source

A *floating source* is a source in which the voltage signal is not referenced to any common ground, such as earth or building ground. Some common examples of floating signal sources are batteries, thermocouples, transformers, and isolation amplifiers. Notice, as shown in Figure 10.17, that neither terminal of the source is connected to the electrical outlet ground. Thus, each terminal is independent of the system ground.

Figure 10.17. Floating signal source



Measuring Differences

To measure your signal, you can almost always configure your DAQ device to make measurements that fall into one of these three categories:

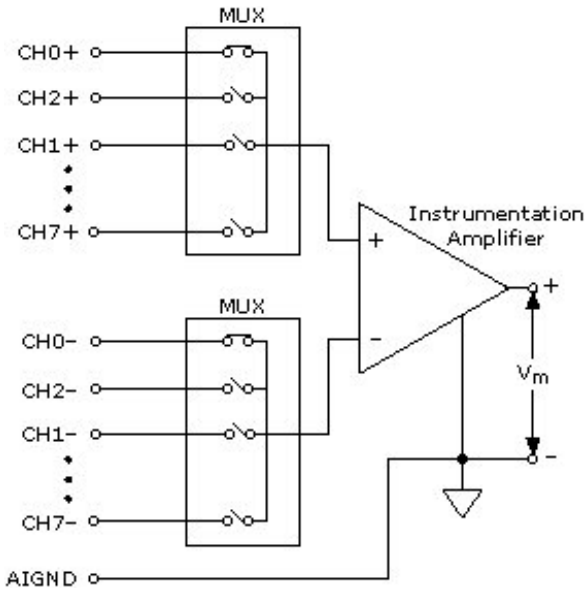
- Differential
- Referenced single-ended
- Nonreferenced single-ended

Differential Measurement System

In a differential measurement system, neither input is connected to a fixed reference such as earth or building ground. Most DAQ devices with instrumentation amplifiers[1] can be configured as differential measurement systems. Figure 10.18 depicts the eight-channel differential measurement system used in the E-series DAQ devices. Analog multiplexers increase the number of measurement channels while still using a single instrumentation amplifier. For this device, the pin labeled AIGND (the analog input ground) is the measurement system ground, as shown in Figure 10.18.

[1] An *instrumentation amplifier* is a special kind of circuit (usually embedded in a chip) whose output voltage with respect to ground is proportional to the difference between the voltages at its two inputs.

Figure 10.18. Differential measurement system

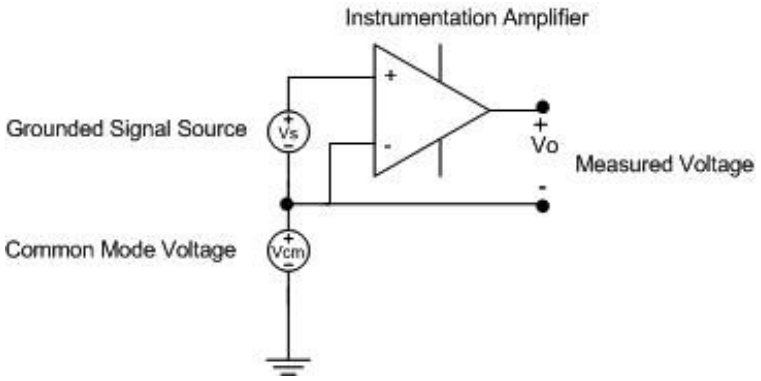


Before we discuss single-ended systems, it is worth noting that SCXI systems always use the popular differential measurement system, while most plug-in DAQ devices give you a choice.

For True Geeks Only

An ideal differential measurement system reads only the potential *difference* between its two terminalsthe (+) and (-) inputs (see Figure 10.19). Any voltage present at the instrumentation amplifier inputs with respect to the amplifier ground is referred to as a *common-mode voltage*. An ideal differential measurement system completely rejects (does not measure) common-mode voltage. Practical devices, however, limit this ability to reject the common-mode voltage. The common-mode voltage range limits the allowable voltage swing on each input with respect to the measurement system ground. Violating this constraint results not only in measurement error but also in possible damage to components on the DAQ device. The common-mode voltage rejection ratio quantifies the ability of a DAQ device, operating in differential mode, to reject the common-mode voltage signal.

Figure 10.19. Ideal differential measurement system



You measure the common-mode voltage, V_{cm}, with respect to the DAQ device ground, and calculate it using the following formula:

$$V_{cm} = \frac{V^+ + V^-}{2}$$

where

V+ = Voltage at the *noninverting* terminal of the measurement system with respect to the instrumentation amplifier ground.

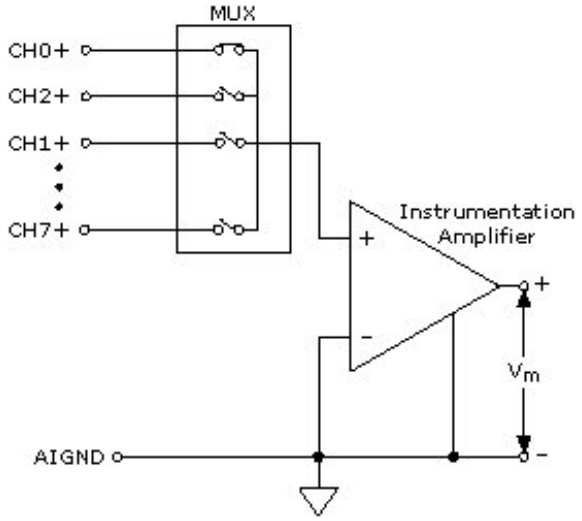
V- = Voltage at the *inverting* terminal of the measurement system with respect to the instrumentation amplifier ground.

You cannot have an arbitrarily high common-mode voltage when measuring with a DAQ device. All plug-in DAQ devices specify a maximum working voltage (MWV) that is the maximum common-mode voltage the DAQ device can tolerate and still make accurate measurements.

Referenced Single-Ended Measurement System

A *referenced single-ended* (RSE) measurement system, also called a grounded measurement system, is similar to a grounded signal source, in that the measurement is made with respect to earth ground. Figure 10.20 depicts a 16-channel RSE measurement system.

Figure 10.20. Referenced single-ended (RSE) measurement system

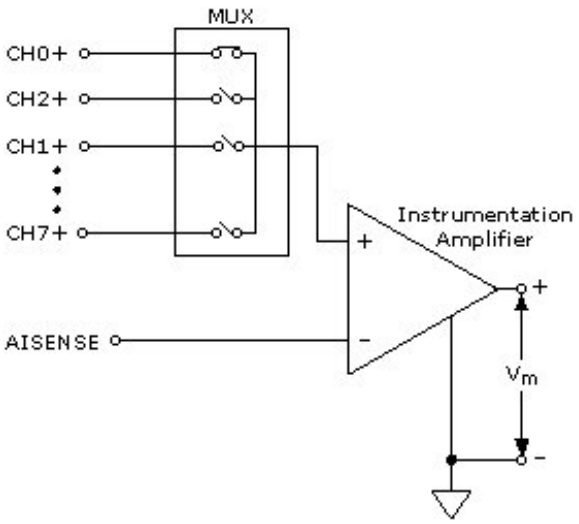


Be careful when connecting a voltage or current source to your DAQ device. Make sure the signal source will not exceed the maximum voltage or current that the DAQ device can handle. Otherwise, you could damage both the DAQ device and your computer.

NRSE Measurement System

DAQ devices often use a variant of the RSE measurement technique, known as the *nonreferenced single-ended* (NRSE) measurement system. In an NRSE measurement system, all measurements are made with respect to a common reference ground, but the voltage at this reference can vary with respect to the measurement system ground. Figure 10.21 depicts an NRSE measurement system where AISENSE is the common reference for taking measurements and AIGND is the system ground.

Figure 10.21. Nonreferenced single-ended (NRSE) measurement system



Incidentally, your measurement system is determined by how you configure your DAQ device. Most devices from National Instruments can be configured for differential, RSE, or NRSE from a software utility (called NI-MAX on Windows platforms). Some of their older devices also have to be configured at the board by placing jumpers in certain position. When you configure a particular DAQ device for a particular measurement system type, all your input channels will follow that measurement type. You should note that you can't change this from LabVIEWyou have to decide ahead of time what kind of measurement you're making.

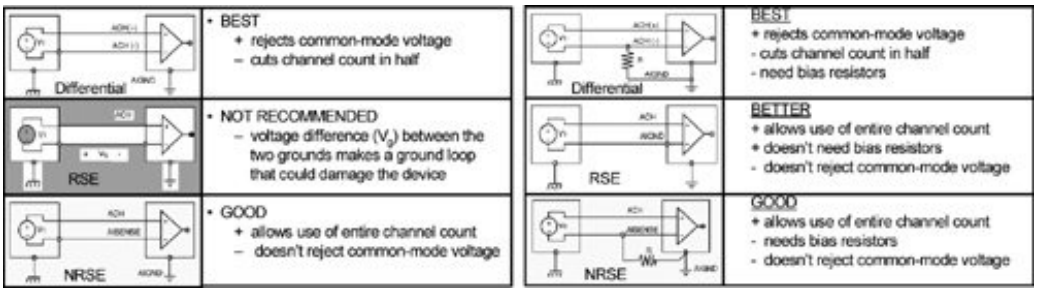
The general guideline for deciding which measurement system to pick is to measure grounded signal sources with a differential or NRSE system, and floating sources with an RSE system. The hazard of using an RSE system with a grounded signal source is the introduction of *ground loops*, a possible source of measurement error. Similarly, using a differential or NRSE system to measure a floating source will very likely be plagued by *bias currents*, which cause the input voltage to drift out of the range of the DAQ device (although you can correct this problem by placing bias resistors from the inputs to ground).

Figure 10.22 summarizes the measurement configurations for each signal type.

Grounded Signal Sources	Floating Signal Sources
Examples: instruments with nonisolated outputs	Examples: thermocouples, battery devices, signal conditioning with isolated outputs

Figure 10.22. Measurement configurations for each signal type

[View full size image]



Sampling, Aliasing, and Mr. Nyquist

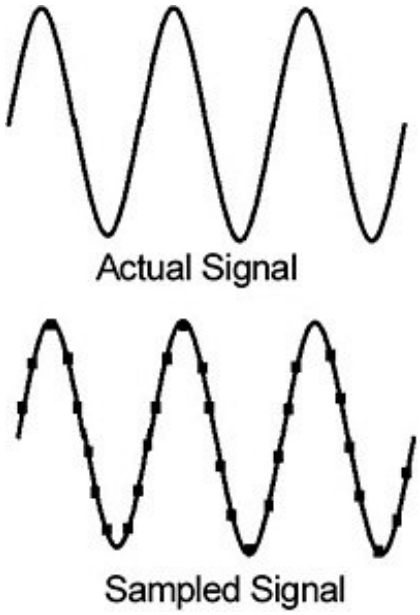
The last, and perhaps most important, part of signal theory we're going to cover here is sampling.

Zeno's arrow[2] aside, real-world signals are continuous things. To represent these signals in your computer, the DAQ device has to check the level of the signal every so often and assign that level a discrete number that the computer will accept; this is called an analog-to-digital conversion. The computer then sort of "connects the dots" and, hopefully, gives you something that looks similar to the real-world signal (that's why we say it *represents* the signal).

> [2] An ancient Greek paradox that relates to whether time is continuous or discrete. See http://en.wikipedia.org/wiki/Zeno's_paradoxes#The_arrow_paradox, for more information about the arrow paradox and to learn about more of Zeno's paradoxes.

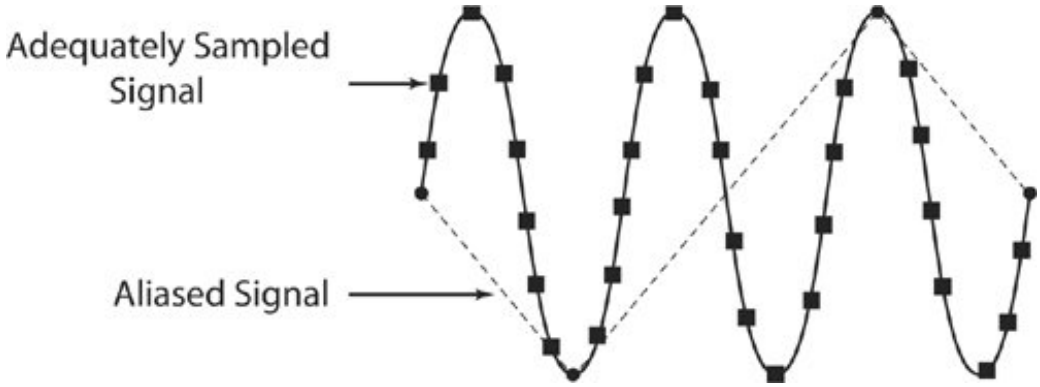
The *sampling rate* of a system simply reflects how often an analog-to-digital conversion (ADC) takes place. Each data point (represented by a square) on the sampled signal in Figure 10.23 represents one analog-to-digital conversion. If the DAQ system is making one ADC every half a second, we say the sampling rate is 2 samples/second, or 2 Hz. Alternatively, we can specify the sampling period, which is the inverse of the sampling rate (in this example, 0.5 sec). It turns out that the sampling rate has a terribly important effect on whether your digitized signal looks anything like the real-world signal.

Figure 10.23. An actual signal and a sampled signal with dots representing digitized/sampled values



When the sampling rate isn't high enough, a scary thing happens. Aliasing, while not intuitive, is easy to observe (see Figure 10.24). If we sample 8.5 times slower (the circles), our reconstructed signal looks nothing like the original.

Figure 10.24. Signal alias caused by inadequate sampling



Aliasing has the effect of introducing frequencies into your data that didn't exist in the real-world signal (and removing some that did), thereby severely distorting your signal. Once you have aliased data, you can never go back: There is no way to remove the "aliases." That's why it's so important to sample at a high-enough rate.

How do you determine what your sampling rate should be? A guy named Nyquist figured it out, and his principle, called Nyquist's Theorem,[3] is simple to state:

[3] More formally know as the Nyquist-Shannon Sampling Theorem. Nyquist formulated it; Shannon formally proved it.

> To avoid aliasing, the sampling rate must be greater than twice the maximum frequency component in the signal to be acquired.

So, for example, if you know that the signal you are measuring is capable of varying as much as 1000 times per second (1000 Hz), you'd need to choose a sampling rate higher than 2 kHz. Notice that the whole Nyquist Sampling Theorem implies that you know what the highest frequency component will be. It is imperative that you find out if you don't know already; if you can't know ahead of time what the highest frequency component will be, you'll need to filter the signal to remove potential high-frequency components, as we describe next.

The Nyquist Theorem only deals with accurately representing the frequency of the signal. It doesn't say anything about accurately representing the shape of your signal. To adequately preserve the shape of your signal, you should sample at a much higher rate than the Nyquist frequency*, generally at least 5 or 10 times the maximum frequency component of your signal*.

Another reason for knowing the frequency range of your signal is *anti-aliasing filters* (lowpass filters). In many real-world applications, signals pick up a great deal of high-frequency noise, glitches, or spikes that will greatly exceed the theoretical frequency limit of the frequency measurement you are making. For example, a common biomedical signal is the electrocardiogram (ECG or EKG), a voltage that is related to heart activity. Although these signals rarely have components beyond 250 Hz, the electrode leads easily pick up RF (radio-frequency) noise in the 100 kHz and MHz range! Rather than sample at extremely high frequencies, these DAQ systems implement some low-pass filters that cut out waveforms above the 250 Hz. The DAQ device can then breathe easier and sample at only, say, 600 Hz.

The only case where sampling rate is not important is in the so-called DC signals, such as temperature or pressure. The physical nature of these signals is such that they cannot vary by much in less than a second. In these cases, a low sampling rate like 10 Hz should do.

In Conclusion . . .

We've covered a lot of issues involving the path from the physical phenomena to the DAQ devices. If you didn't grasp many or even most of these concepts at first, don't worry. DAQ theory is a complex subject, and unless you've had some experience in this area of electrical engineering, it can take some practice before you understand it all.

You've seen a summary of how signals are classified, what kind of transducers are often used, the importance of signal conditioning, the different measurement configurations for digitizing grounded or floating signal sources, and the necessity of using Nyquist's Sampling Theorem. You'd need to take a

couple of electrical engineering courses to thoroughly cover the whole topic of data acquisition and instrumentation; we've just skimmed the surface in this section. Nonetheless, it should be enough to get you started with your measurements.

PREV NEXT

Selecting and Configuring DAQ Measurement Hardware

Choosing Your Hardware

Once you know what kind of signals you want to measure and/or generate, it's time to choose a plug-in DAQ device (assuming a DAQ device will meet your requirements), such as those shown in Figure 10.25. Generally speaking, we recommend always using a National Instruments device if you're going to use LabVIEW. National Instruments offers a huge selection of all types of DAQ devices with a good selection of buses, platforms, performance, functionality, and price range. You can browse their catalog online (http://ni.com), where you will find a "DAQ Designer" utility that will help you determine what hardware suits your needs. If you do find a DAQ device from a company other than National Instruments, make sure they provide a LabVIEW driver with it.

Figure 10.25. DAQ devices in a variety of form factors and bus types

[View full size image]

To pick the best hardware for your system, you need to understand well what your system requirements are, most noticeably the "I/O count" (how many inputs and how many outputs). The following checklist should be useful in determining if you have all the information you need to select a DAQ device:

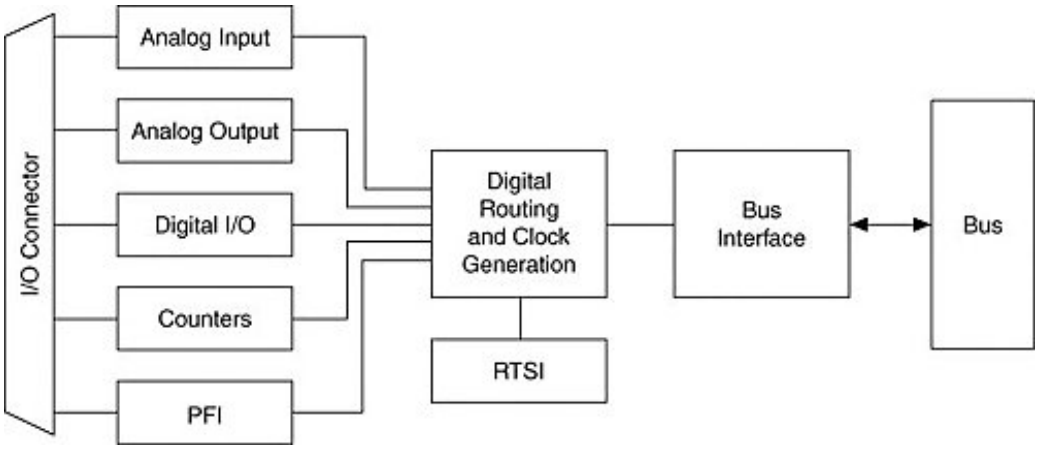
- What type of operating system am I using (Windows, Linux, Mac OS X)?
- What type of bus or connector is available (PCI, PC-Card for laptops, USB, etc.)?
- How many analog inputs will I need (multiply by 2 if you need differential inputs)?
- How many analog outputs will I need?
- Are the analog inputs voltage, current, or both? What are the ranges?
- How many digital input and output lines will I need?

- Do I need any counting or timing signals? How many?
- Do any of the analog I/O signals require special signal conditioning (e.g., temperature, pressure, or strain)?
- Will any of the analog I/O signals exceed 10 V or 20 mA?
- What is the minimum sampling rate required on any one channel?
- What is the minimum scan rate for all the channels?
- What precision or resolution (12-bit, 16-bit, or 24-bit) will I need?
- Is portability, ruggedness, isolation, or cost an issue? If so, what are the trade-offs?
- Have I accounted for my needs in the future (expansion, new DAQ systems, etc.)?

By answering these questions, you'll be in a better position to choose a suitable DAQ device. At the time of this writing, a popular type of National Instruments device is the so-called *M Series* device, which is a "multi-function" device.

A multifunction DAQ device like an M Series device will provide you with analog inputs, analog outputs, digital I/O, and often at least one counter/timer, making it useful for a wide range of applications (see Figure 10.26).

Figure 10.26. Multi-function M Series DAQ device from National Instruments



The analog input section of a multifunction device consists of the ADC and the *analog input circuitry*. The analog input circuitry contains the analog multiplexer, the instrumentation amplifier, and the sample-and-hold circuitry. For more detailed information on the ADC, refer to the manual that accompanies your DAQ device.

What about cost? Plug-in DAQ devices range in price from about $100 (U.S.) for low-cost USB

devices to $5000 (U.S.) (or more) for extremely high-end devices. In general, three factors are directly proportional to and most influential on the price: the communication bus, the sampling rate, and the number of analog I/O channels. So if you are making DC measurements, there's no need to shell out extra dough for a device with a 10-MHz sampling rate. Also, digital I/O is usually cheap. The device prices shouldn't really seem expensive once you pause to consider what you're getting, however. How much would an oscilloscope, spectrum analyzer, strain gauge meter, and hundreds of other instruments all together cost you if you had to buy them as "non-virtual" instruments?

Don't forget: One of the best resources to help you figure out what hardware you need is the online utility from National Instruments called DAQ Designer. Look for it on http://ni.com. If you want to look at a broader range of options, you should consult with an experienced system integration company who can advise you on what works best for you.

Activity 10-2: Measurement System Analysis

Following are a couple of more challenging signal measurement problems. Your objective is to specify the needed information.

Answer the following for each scenario:

1. What kind of signals need to be measured?
2. What signal measurement type do you recommend?
3. What should the sampling rate be for their signals? What is the Nyquist frequency?
4. Is any signal conditioning needed? If so, what?
5. What hardware would you pick for this system?

A. Professor Harry Phace, of the biomedical engineering lab, wants to acquire heart signals from human subjectshopefully without electrocuting them. He wants to measure electrocardiograms from two subjects at a time. Each subject has four electrodes connected to his body at different places. The objective is to measure in real time the potential between each of three electrodes, with the fourth electrode designated as a reference. The leads from the electrodes to the DAQ system have no isolation or ground shields. The maximum amount of detail expected in the waveforms are segments 2 ms wide. The signals are within a 0.024 mV range.

B. Ms. I. M. Aynurd needs to measure how the resistance of a flexible material changes under stress and high temperatures. To do so, she has a special chamber with a machine that repetitively twists and stretches the material. The chamber also functions as an oven with a variable temperature control. She wants to observe in real time (within 1/10 sec) how the resistance of each of 48 strands of this material changes inside the chamber. The resistance is measured by applying a known voltage to each strand and measuring the current. The temperature of the chamber is monitored through a thermocouple. The stress machine is turned on and off through a solid-state relay. Finally, the number of cycles from the stress machine needs to be measured.

The answers are at the end of the chapter.

Installing DAQ Device Driver Software

All devices that your computer communicates with use *drivers*, nasty and painful pieces of low-level code that convince your computer that the devices are connected and can be used. The good news is that, assuming your drivers are installed properly, you should not have to really mess with them to use your DAQ device. All National Instruments DAQ devices come with driver software, collectively referred to as NI-DAQmx. In fact, NI-DAQmx is installed by default when you install LabVIEW, so if you have the full version of LabVIEW, chances are it's already on your machine.

Between NI-DAQmx and LabVIEW, there is a utility called MAX (Measurement & Automation Explorer). MAX is a Windows software interface that gives you access to all your National Instruments devices (whether they are DAQ, GPIB, VXI, and so on). MAX is mainly useful for configuring and testing your hardware. This is very useful to do before you try to access the hardware in LabVIEW. Again, MAX is installed by default when you installed LabVIEW; you should see the shortcut icon on your Windows desktop (see Figure 10.27).

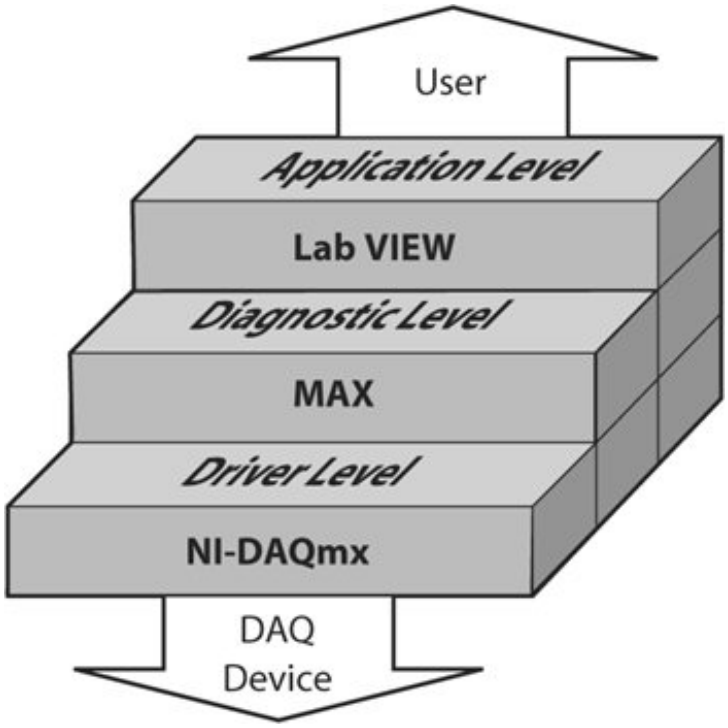
Figure 10.27. Measurement & Automation shortcut that launches MAX (on Windows)

We'll talk more about MAX briefly.

MAX is a utility available on Windows only. For configuring DAQ devices on other operating systems, such as the Mac OS X or Linux, use the NI-DAQmx or NI-DAQmx Base configuration utility provided for that OS.

Figure 10.28 shows the conceptual relationship between an NI-DAQmx Device, NI-DAQmx, MAX, LabVIEW, and the user.

Figure 10.28. The conceptual relationships connecting the user to the DAQ Device

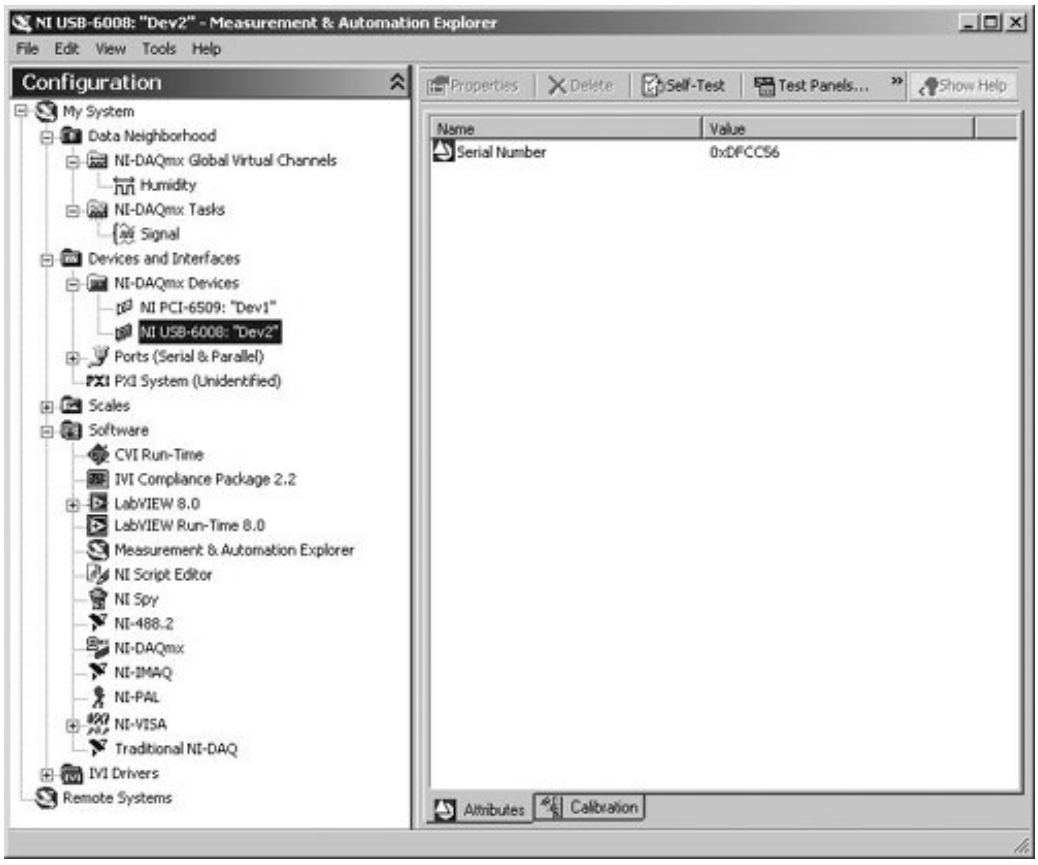


Measurement & Automation Explorer (MAX)

MAX, shown in Figure 10.29, is mainly used to configure and test your National Instruments hardware, but it does offer other functionality such as checking to see if you have the latest version of NI-DAQmx installed.

Figure 10.29. The MAX configuration utility

[View full size image]



If you are using Mac OS X or Linux, the previous section on MAX doesn't apply, because, at least at press time, MAX was a Windows-only utility. However, NI-DAQmx and NI-DAQmx Base have driver configuration utilities for the supported platformsoften these are command-line tools. You should check the documentation for information on installing and configuring your hardware.

Next, we will show you how to configure your DAQ devices to quickly take useful measurementswe will do all of this from within MAX.

NI-DAQmx

NI-DAQmx is a cross-platform driver for National Instruments DAQ devices. NI-DAQmx supersedes the older Traditional NI-DAQ (previously referred to as "NI-DAQ") and provides the following improvements over Traditional NI-DAQ:

- Improved state model
- Multithreaded driver
- Robustness in exceptional conditions
- Simplified synchronization
- Decreased LabVIEW diagram clutter
- Smooth transition from easy to advanced programming

Bottom-line: It makes your life easier by helping you achieve your DAQ objectives with less work and better results!

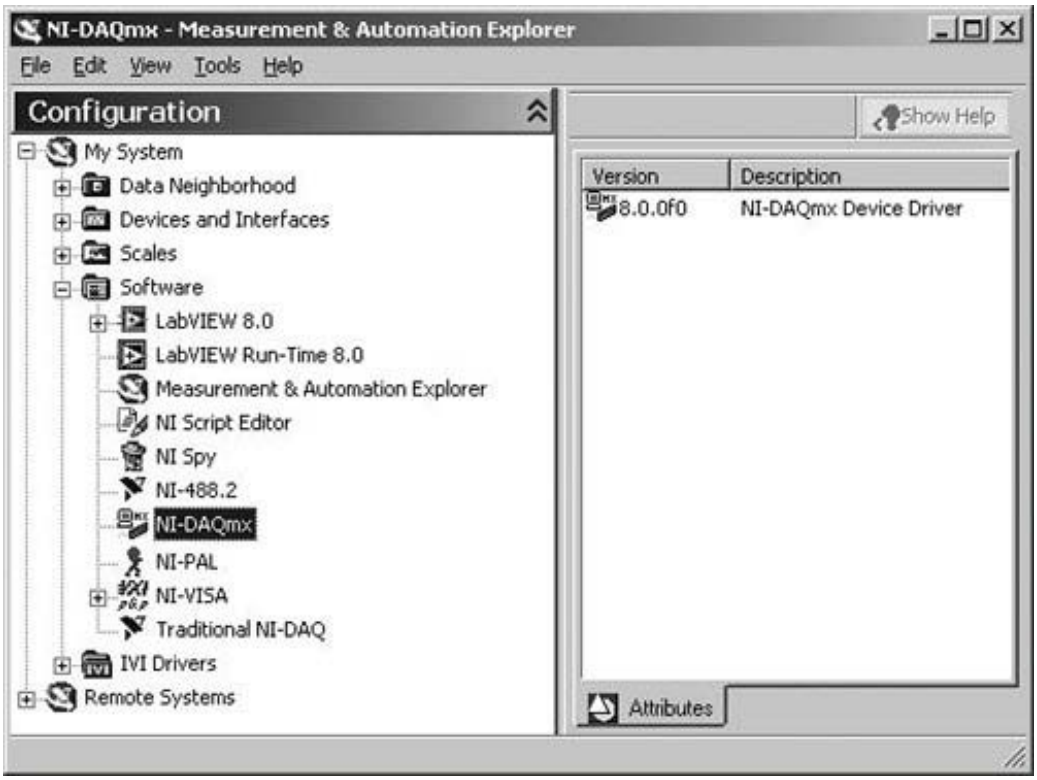
At the time of this book's writing, NI-DAQmx is supported on Windows and Linux. It seems likely that Mac OS X will also be supported some time in the future. If you would like to use National Instruments DAQ devices on platforms that are not currently supported by NI-DAQmx, consider using NI-DAQmx Base, which is supported on Windows, Linux, Mac OS X, and Pocket PC.

NI-DAQmx Base offers only a subset of NI-DAQmx functionality on Windows, Linux, Mac OS X, and Pocket PC OSs. The LabVIEW VIs provided with NI-DAQmx Base are similar to the NI-DAQmx VIs, but they are not interchangeable. Interestingly, the NI-DAQmx Base driver software was built almost entirely in LabVIEW, using register-level programming via the NI Measurement Hardware DDK (MHDDK). This architecture makes it easy to port to new platforms in the future. It will be a long time, if it ever happens, before NI-DAQmx Base provides all of the functionality of NI-DAQmx. But, as LabVIEW continues to evolve as a full-featured software development environment, this possibility is not out of the question.

Before we move to the next sections, make sure that DAQmx is installed on your system. Do this by browsing to My System>>Software in the MAX Configuration panel. Find NI-DAQmx and mouse-click on it to select it. This will display the NI-DAQmx software version information in the Attributes pane, as shown in Figure 10.30.

Figure 10.30. Verifying that NI-DAQmx is installed from within MAX

[View full size image]



The MAX Software category shows all of your currently installed versions of National Instruments software. The icon for each software package is also a shortcut that you can use to launch the software. For example, you can right-click on the LabVIEW 8.0 icon and choose Launch LabVIEW 8.0 to launch LabVIEW. The Software category also includes a Software Update Wizard to check if your National Instruments software is the latest version. Right-click on the Software category icon and select Get Software Updates to run the Software Update Wizard (see Figure 10.31). If your software isn't the latest version, the Software Update Wizard will link you to a page on ni.com to download the latest version of your software.

Figure 10.31. Checking for software updates in MAX

Configuring NI-DAQmx Devices in MAX

Now we are now going to show you how to configure your NI-DAQmx devices. If you don't have an NI-DAQmx device connected to your computer, you should know that NI-DAQmx supports simulated devices. So, if you don't have an NI-DAQmx device, jump ahead to the next section and add a simulated device. Then, come back here and we'll show you how to configure it just like you

would a real device!

Browse to the My System>>Devices and Interfaces>>NI-DAQmx Devices category in MAX. Beneath it you should see a list of all the physical and simulated NI-DAQmx devices connected to your system (as shown in Figure 10.32).

Figure 10.32. NI-DAQmx devices in MAX, showing the physical and simulated devices connected to your system

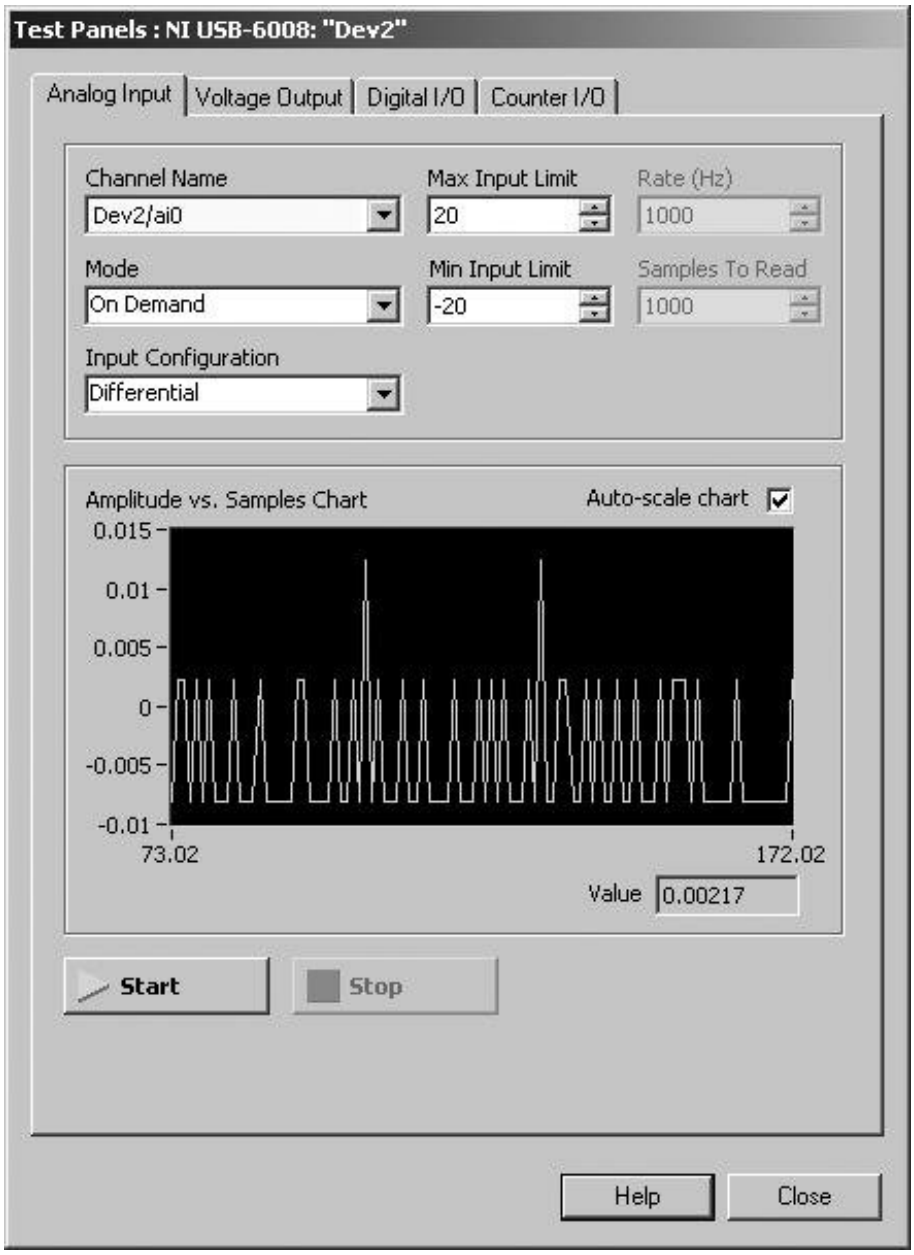
In addition to NI-DAQmx devices, the Devices and Interfaces category shows you any currently installed and detected National Instruments hardware, such as plug-in DAQ devices, SCXI and PXI chasses, GPIB controllers, and so on. Devices and Interfaces also includes utilities for configuring and testing your devices. We aren't going to show you how to configure every type of device. But hopefully, this section will give you enough experience to explore the rest of MAX comfortably.

When you right-click on an installed NI-DAQmx device, the pop-up menu gives you the options: Self-Test, Test Panels, Reset Device, Create Task, Configure TEDS, Rename, Delete, Device Pinouts, Properties, and Self-Calibrate.

The Self-Test option runs a brief test of device resources, and then presents a message window with the results of the test.

The Test Panels window (shown in Figure 10.33) is used for testing the analog input, analog output, digital I/O, and counter functionality of your DAQ device. This is a great utility for troubleshooting because it allows you to test the functionality of your device directly from NI-DAQmx. If your device doesn't work in the Test Panel, it isn't going to work in LabVIEW. If you are ever having unexplainable trouble with data acquisition in a LabVIEW program, you can use the Self-Test and the Test Panels to make sure the device is working properly.

Figure 10.33. Test Panels window



The Reset Device option performs a hardware reboot of the device. A reset aborts any running tasks and restores the device to its default settings.

The Create Task option creates an NI-DAQmx task, which you will learn about later in this section.

The Configure TEDS option allows you to configure TEDS sensors connected to your DAQ device. TEDS stands for Transducer Electronic Data Sheets, and is an IEEE standard (IEEE 1451.4) for smart sensors that store their own calibration data and can communicate that information to the computer that is acquiring the electrical signal from the sensor. For more information on TEDS, visit

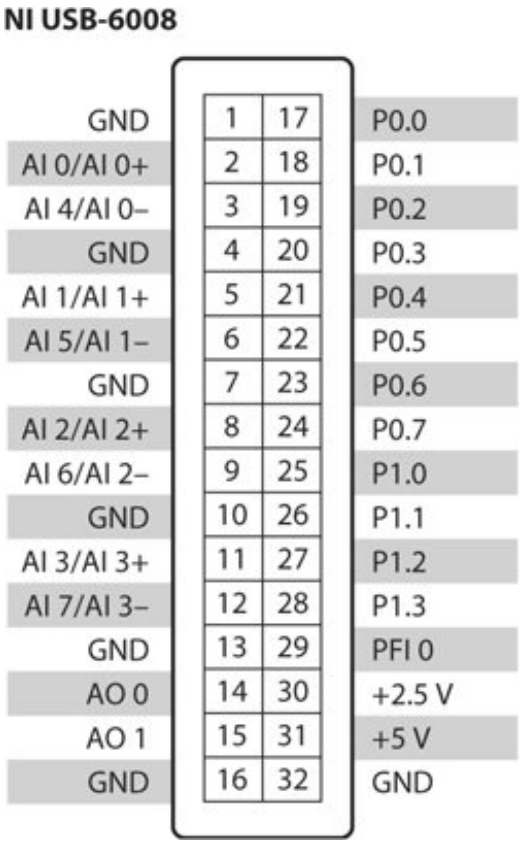
http://www.ni.com/teds/.

The Rename option allows you to change the name that you will use to address the device in LabVIEW. The names default to "Dev1," "Dev2," and so on.

The Delete option will be disabled for plug & play physical devices, but will be enabled for simulated devices and accessories.

The Device Pinouts option will open the NI-DAQmx Device Terminal help documentation, showing you the pin numbers for connecting physical signals to and from your DAQ device (see Figure 10.34).

Figure 10.34. Device pinouts, as shown by the NI-DAQmx Device Terminal help documentation



The Properties panel allows you to configure additional device-specific options such as power-up states, accessory, RTSI configuration, and so on.

The Self-Calibrate option runs a self-calibration on devices that support calibration.

NI-DAQmx Simulated Devices

Even if you do not have a DAQ device present on your system, don't let that stop you from learning how to configure and use DAQ devices in MAX, or how to program your DAQ application. NI-DAQmx allows you to add simulated devices to your system, configure and test them in MAX, and acquire data from them in LabVIEWit works just as if you had real hardware present! And, when you are ready to put the real hardware into your system, you can easily import the NI-DAQmx simulated device configuration to the physical device.

To create an NI-DAQmx simulated device, complete the following steps within MAX:

1. Right-click NI-DAQmx Devices (found beneath Devices and Interfaces) and select Create New NI DAQmx Device...>>NI-DAQmx Simulated Device from the pop-up menu, as shown in Figure 10.35.

Figure 10.35. Creating a new NI-DAQmx Simulated Device in MAX

[View full size image]

2. In the Choose Device dialog box (see Figure 10.36), use the tree control to browse to the device you want to simulate and then press the OK button. Devices are organized by family.

Figure 10.36. Choose Device dialog, which allows you to specify the simulated device type to create

If you are not sure which type of simulated device to create, choose a general purpose device with lots of different I/O. For example, the M Series 6221DAQ device has 16 analog inputs, two analog outputs, 24 digital I/O lines, and two counter/timersthis will allow you to test just about any type of data acquisition task.

In MAX, the icons for NI-DAQmx simulated devices are yellow. The icons for physical (real) devices are green.

To remove an NI-DAQmx simulated device, right-click the device and click Delete from the pop-up menu.

Configuring Data Acquisition

We will now discuss how to configure NI-DAQmx Scales, NI-DAQmx Virtual Channels, and NI-DAQmx Tasks in MAXthese are the key components to getting useful measurements into LabVIEW. A scale defines scaling information that can be used by virtual channels, virtual channels define real-world measurements consisting of one or more DAQ channels, and tasks define timing, triggering, and sample size information for acquiring data from virtual channels.

NI-DAQmx Scales

A scale defines scaling information that can be used by virtual channels (see the Custom Scaling setting in the virtual channel's configuration panel). This is sometimes necessary or useful particularly for sensors that are not linear or where you want to read the actual units directly instead of having to convert voltage or current to the desired unit, such as temperature. Each custom scale can have its own name and description to help you identify it (see Figure 10.37). A custom scale can be one of four types: linear, map ranges, polynomial, or table, described as follows:

- Linear A scale that uses the formula `y = mx + b`.
- Map ranges A linear scale where the user enters two XY coordinate pairs instead of entering values for `m` and `b`.
- Polynomial A scale that uses the formula `y = a0 + a1*x + a2*x2 + ... + an*xn`.
- Table A scale where you enter the raw value and the corresponding scaled value in a table

format.

Figure 10.37. NI-DAQmx Scales in MAX

NI-DAQmx Virtual Channels

So what is a virtual channel? Virtual channels define real-world measurements consisting of one or more DAQ channels (terminals on your DAQ device) along with other channel-specific informationrange, terminal configuration, and custom scalingthat is used to format the data. Most DAQ devices simply measure or generate electrical voltages or currents (analog voltages between -10V and +10V, analog currents between 20mA and 40 mA, digital voltages that are either 0V or +5V, and so on). However, when we want to measure physical phenomena like temperature, humidity, and wind-speed, we have to convert one or more electrical signals, as measured by our DAQ devices, into real-world measurements. This is what virtual channels do and why they are so powerful.

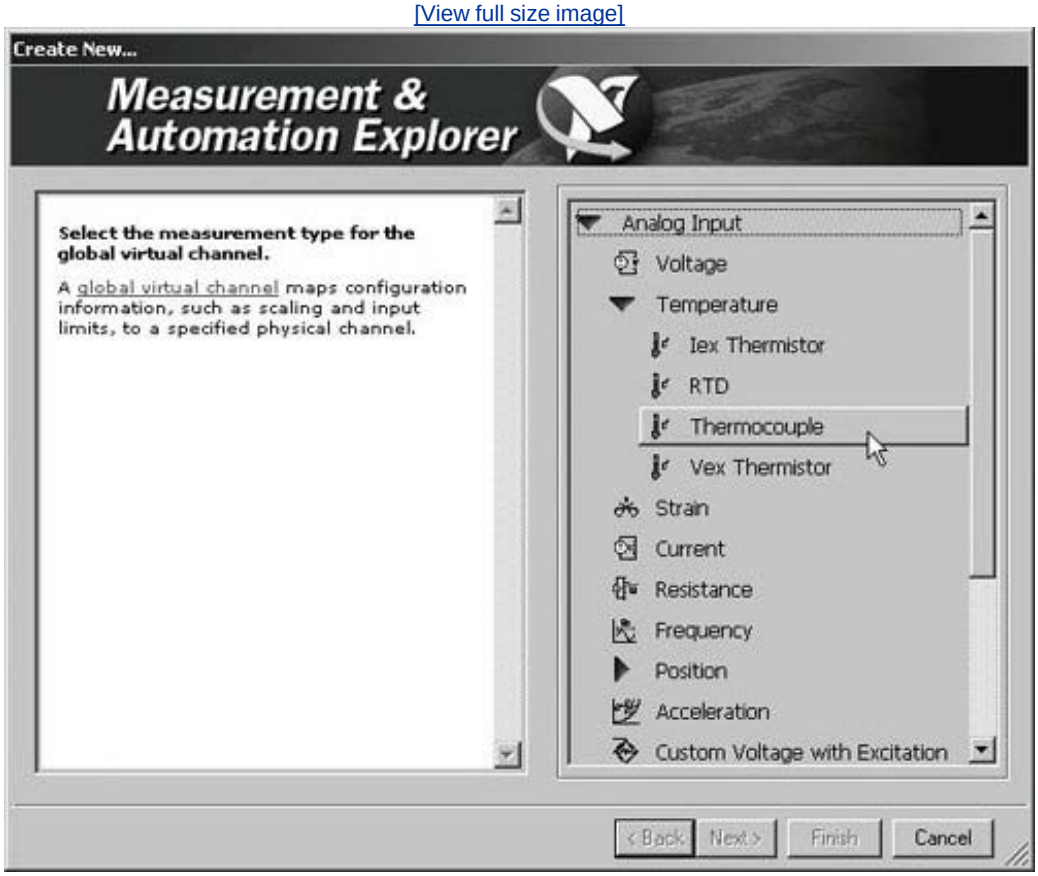
You don't *have* to use virtual channels, of course. In your LabVIEW application, you can refer to device channels by their number (0, 1, 2, ...). But configuring them first as a virtual channel in MAX is handy because, as we'll see in the next chapter, you can use a LabVIEW front panel or block diagram DAQmx Global Channel ring that gives you the names of all your virtual channels, and you can even automatically generate LabVIEW VIs from your DAQmx Global Channel (see Figure 10.38)!

Figure 10.38. Assigning a DAQmx Global Channel to a DAQmx Global Channel ring on the front panel of your VI

To create a virtual channel, you right-click on the Data Neighborhood icon in MAX and choose Create New . . . . Select NI-DAQmx Global Virtual Channel and click Next. This will guide you through the steps to set up your virtual channel (see Figure 10.39).

Figure 10.39. Create New . . . dialog

[View full size image]



Voltage Input and Output Settings

When configuring virtual channels that use analog I/O, it is important to understand some of the parameters that control the operation of the ADC (analog-to-digital converter) and DAC (digital-to-analog converter). On almost all devices, you will configure these settings in the virtual channel or task settings panel (some very old boards require you to set jumpers on the board). The types of settings you can configure include parameters like range, input mode, reference, and polarity; for example, on most M Series devices, you can set the following:

- ADC Input Range
  - Unipolar 0 V to + 10 V
  - Bipolar ±5 V
  - Bipolar ±10V (default)
- ADC Input Mode
  - Ground-referenced single-ended
  - Nonreferenced single-ended
  - Differential (default)
- DAC Reference
  - Internal (default)
  - External
- DAC Polarity
  - Unipolarstraight binary mode
  - Bipolartwo's complement mode (default)

NI-DAQmx Tasks

Sometimes when you perform a measurement, you will need to coordinate the reading of data from one or more channels with timing and triggering events. An NI-DAQmx Task is a collection of one or more virtual channels along with timing, triggering, and other properties. Conceptually, a task represents a measurement or generation you want to perform. For example, a task allows you to specify whether you want to measure 1 sample, measure N samples, or measure continuously (using a buffer to store data). A task also allows you to specify the sample rate, the timing clock source, and the task triggers. Once you have defined a task, you can simply start the task, read the task data, and stop the task from within LabVIEW.

PREV NEXT

Wrap It Up!

Whew! This chapter has been a heavy oneif you read it end-to-end, you deserve to take a break.

We've covered basic signal and data acquisition theory. Different types of signal can be classified for measurement purposes into analog AC, analog DC, digital on/off, digital counter/pulse, and frequency. Signal sources can be *grounded or floating*. *Grounded signals* usually come from devices that are plugged in or somehow connected to the building ground. *Floating source* examples include many types of sensors, such as thermocouples or accelerometers. Depending on the signal source type, signal characteristics, and number of signals, three measurement systems can be used: *differential*, *referenced single-ended (RSE)*, and *nonreferenced single-ended (NRSE)*. The big no-no is a grounded source using a referenced single-ended system. The sampling rate of a data acquisition system (for AC signals) is very important. According to the *Nyquist Theorem*, the sampling rate must be more than twice the maximum frequency component of the signal being measured.

Selecting a DAQ device or system that will do the right job is the next step. Many kinds of DAQ devices exist for different platforms, applications, and budgets. National Instruments' SCXI and PXI systems provide ways to work with a very high number of channels, as well as performing elaborate *signal conditioning*. Installing a DAQ device is getting easier than before, but still requires a little knowledge of how to set parameters in the configuration utility NI-MAX. The collection of software drivers for National Instruments' devices is called NI-DAQmx, which NI-MAX relies upon to function properly.

Don't feel bad if much of the material in this chapter eluded you. DAQ is a complex subject and often requires the expertise of an instrumentation engineer. However, for relatively simple DAQ systems, if you are willing to experiment and have a sense of adventure, you can assemble your own DAQ system at your PC. Remember to consult the manuals that come with your device and hardware for more details.

Solutions to Activities

The following is a list of solutions to activities in this chapter:

10-1

A1: 1,3,4,2,5,4,1,2,2,2,1

10-2

A.

1. Analog AC signals, small amplitude, floating.
2. Differential (because of small amplitudes relative to ground).
3. Nyquist frequency = *fn* = 1/(2 *ms)* = 1/(2•10^{-3} *ms*) = 500 Hz. Sample at much more than 1000 Hz, such as 5 kHz.
4. Yes. Amplification (small signal amplitude), isolation (safety), and perhaps anti-aliasing filters (noise) are required.
5. Any M Series device should do.

B.

1. Analog input DC (resistance, thermocouple readings), analog output DC (voltage excitation), digital output on/off (relay to turn on machine), digital input counter (count machine cycles).
2. Referenced single-ended (RSE).
3. All DC measurements. 10 Hz should be a good sampling frequency.
4. Yes, thermocouple conversion.
5. SCXI system, because of high channel count and special conditioning needed.

11. Data Acquisition in LabVIEW

Overview

In this chapter, we'll get to the heart of what LabVIEW is often commissioned to do: data acquisition. You'll become familiar with some of the VIs on the Measurement I/O>> DAQmxData Acquisition palette. Analog input and output, digital input and output, and counter input and output are covered. We'll go through the basic steps necessary to get you started with some data acquisition in LabVIEW, and point you in the right direction to do your own more advanced DAQ and instrument control programs.

Goals

- Investigate the I/O palette on the front panel: waveform datatype, DAQ Channel names
- Become familiar with the basic VIs in the Measurement I/O>>DAQmx palette
- Learn the sequence of VIs that you should use for simple DAQ measurements, both analog and digital
- Get started with some examples and DAQ wizards
- Learn about more advanced DAQ concepts, such as continuous data acquisition and streaming to disk

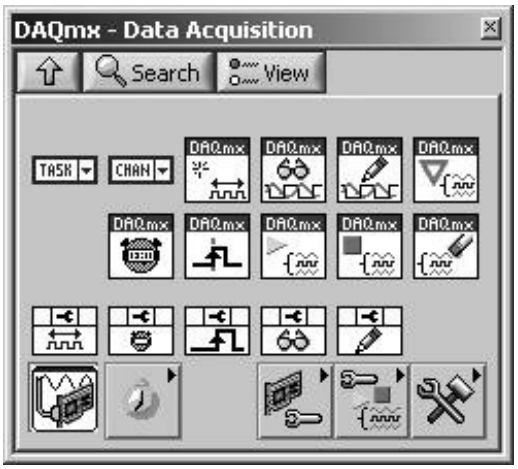
Key Terms

- DAQ Assistant
- Analog input
- Buffer
- Sample
- Sampling rate
- Scale
- Channels
- Virtual channel
- Analog output
- Triggering
- Waveform
- NI-DAQmx Task
- Port
- Line
- Streaming
- Continuous acquisition
- Waveform
- Wizard
- Counter
- Pulse

Understanding Analog and Digital I/O

The Measurement I/O>>DAQmxData Acquisition palette (see Figure 11.1) contains all of the VIs, and other tools that you will need to explore DAQ, or Data Acquisitionone of LabVIEW's great strengths. In fact, LabVIEW's DAQ capabilities might even be the reason why you chose to learn LabVIEWso that you could quickly acquire data and generate signals to measure, control, turn on and off, or blow up stuff in the external world!

Figure 11.1. DAQmxData Acquisition palette



To do this effectively, you first need to understand a little bit about the interface between LabVIEW and the DAQ devices, and some basic concepts about how analog and digital I/O are handled in software.

Before we get into some of this theory, though, let's first see how easy it is to do DAQ in LabVIEW using the DAQ Assistant.

Using the DAQ Assistant

Before we cover analog and digital I/O in depth, it's time to try acquiring data hands-on in LabVIEW first. You can jump right in using the DAQ Assistant. The DAQ Assistant is an Express VI that creates, edits, and runs tasks using NI-DAQmx.

You can find the DAQ Assistant (see Figure 11.2) on the Measurement I/O>>NI-DAQmxData

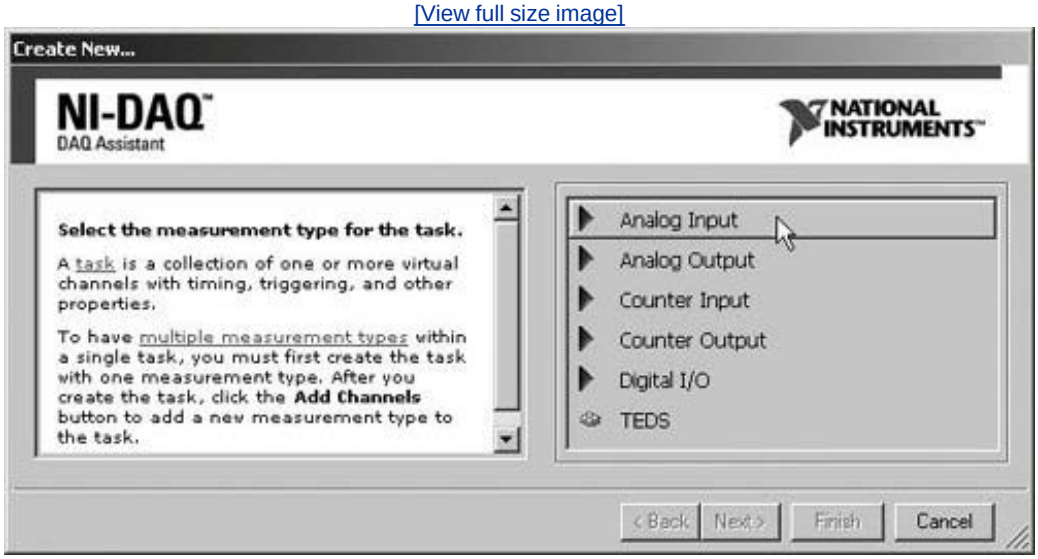
Acquisition palette of the Functions palette.

Figure 11.2. DAQ Assistant express VI

The first time the configuration dialog of this express VI is opened (by placing it onto the block diagram, for example), you will be presented with a dialog that guides you through the process of selecting a signal measurement (input) or generation (output) type (shown in Figure 11.3).

Figure 11.3. DAQ Assistant configuration dialog when creating a new task

[View full size image]



This dialog (see Figure 11.3) is used for creating new NI-DAQmx tasks, and it is accessible from a variety of locations in LabVIEW and MAX. You will learn all about NI-DAQmx tasks in the section, "NI-DAQmx Tasks."

From this dialog, you can choose from any type of signal measurement or generation supported by NI-DAQmx. After you have finished selecting the measurement type and channels, you will be presented with the DAQ Assistant property page for configuring the task.

Once the measurement type for a DAQ Assistant Express VI instance is selected, it cannot be changed. You can, however, add and remove channels of the same measurement type; but if you wish to change measurement type, you will have to start over with a new DAQ Assistant VI.

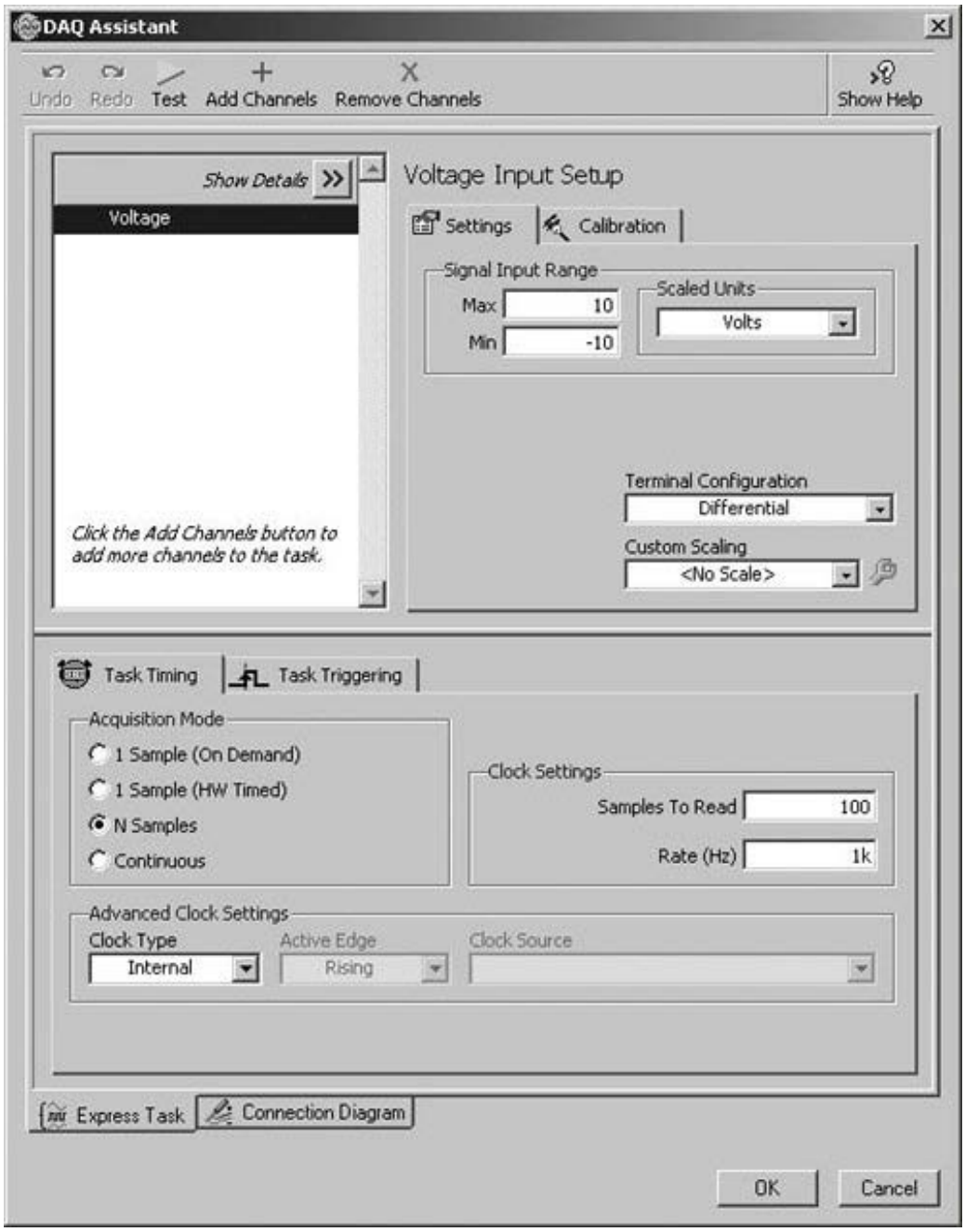
After pressing the OK button, you will see a message showing the progress while LabVIEW builds your DAQ Assistant VI. Once complete, you will see that there is now a "data" terminal for reading or writing the signal, as shown in Figure 11.5. This "data" terminal is a *dynamic data type*, which we learned about in Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs." This can contain scalars and waveforms compatible with any I/O type that the DAQ Assistant supports.

Figure 11.5. DAQ Assistant express VI with "data" terminal visible

Now, let's take a look at some practical examples of using the DAQ Assistant to achieve some common tasks.

Figure 11.4. DAQ Assistant configuration dialog when configuring a task

[View full size image]



Don't have a DAQ system yet? The following activity, and others in this chapter, can be completed (on a computer running Windows) even if you don't have actual DAQ hardware. You can use the NI-DAQmx simulated devices to set up a simulated device that will

generate signals for you. In LabVIEW, the DAQ VIs will behave the same way (although the actual signal may look different, of course).

For help on setting up a simulated device, see the section, "NI-DAQmx Simulated Devices," in Chapter 10, "Signal Measurement and Generation: Data Acquisition."

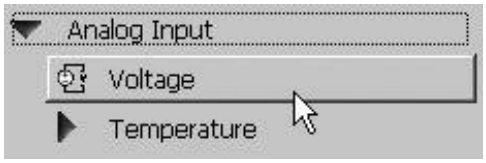
Activity 11-1: Analog Input

1. Connect a voltage source, such as a function generator or a battery, to channel 0 on your DAQ device. Be sure you know whether you've configured your device for differential or single-ended measurements. If you are using a simulated NI-DAQmx device (as described in Chapter 10), there will be a simulated sine wave signal on the analog input channels.

2. Open a new VI, and place a DAQ Assistant VI (Measurement I/O>>NI-DAQmxData Acquisition palette) onto the block diagram.

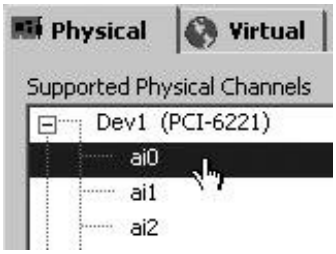
3. From the configuration dialog of your DAQ Assistant VI, choose an Analog Input>>Voltage measurement type (see Figure 11.6).

Figure 11.6. Configuring your task as a voltage analog input type from the DAQ Assistant configuration dialog



4. Select the Physical Channel "ai0," and then press the Finish button (see Figure 11.7).

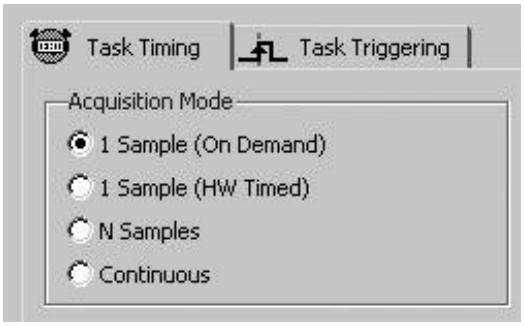
Figure 11.7. Selecting the physical channel for your measurement task



This will complete the process of selecting the measurement type, and you will be presented with the DAQ Assistant task configuration dialog.

5. On the Task Timing tab, select an Acquisition Mode of 1 Sample (On Demand). This will configure your task to return a single measurement value each time we call the DAQ Assistant VI (see Figure 11.8).

Figure 11.8. Configuring the task timing acquisition mode as 1 Sample (On Demand)



6. Press the OK button to close the DAQ Assistant task configuration dialog. LabVIEW will now build your DAQ Assistant VI.

7. Build the front panel and block diagram shown in Figures 11.9 and 11.10.

Figure 11.9. Front panel of the VI you will create during this activity

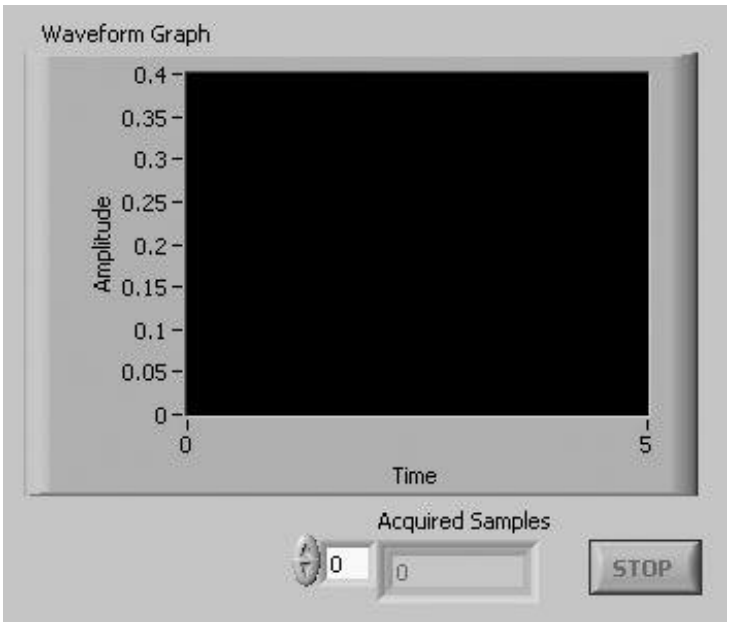
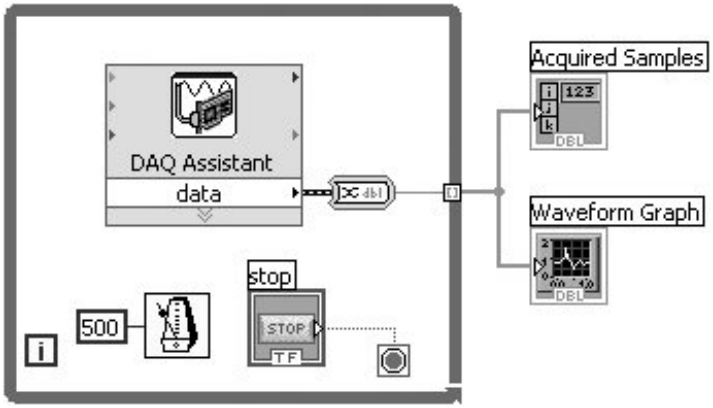


Figure 11.10. Block diagram of the VI you will create during this activity



Convert from Dynamic Data

Convert from Dynamic Data (Express>>Signal Manipulation palette). This Express VI converts the dynamic data type to a numeric scalar. It has been configured (via its Express VI configuration dialog) to return a single scalar floating point number from channel 0.

8. Save your VI as `Quick Analog In.vi`.

9. Run the VI and then press stop after a few seconds.

10. Examine the data in the `Acquired Samples` array.

The While Loop in the previous activity is the trade-off for simplicity: You add unnecessary software overhead. Not a problem for a quick look at your data, but in a decent-sized block diagram, you really don't want the software trying to do all the sampling controlthe DAQ devices and low-level drivers are designed to do that.

This method of DAQ works perfectly fine under many circumstances, provided:

1. The sampling rate is slow (once per second or slower).
2. Other time-consuming operating system (OS) events are not occurring while the VI is running.
3. Slight variations in the sampling times are acceptable.

Suppose you are running a VI using a LabVIEW timing function in a loop, sampling once per second, and you decide to drag the window around. Your VI might not acquire all the data during the drag! Suppose you drag the window for five seconds. When you finally release the mouse button, the VI continues, but it gives no indication that some of the data was not sampled for a five-second period!

We'll talk about more robust methods for acquiring data, such as buffered analog input, in the following sections.

Analog I/O Terms and Definitions

We've already covered some theory of data acquisition and analog I/O in Chapter 10. You learned about sampling rates, scales, virtual channels, and input ranges. If these terms are not familiar to you, you might want to go back and review Chapter 10 again.

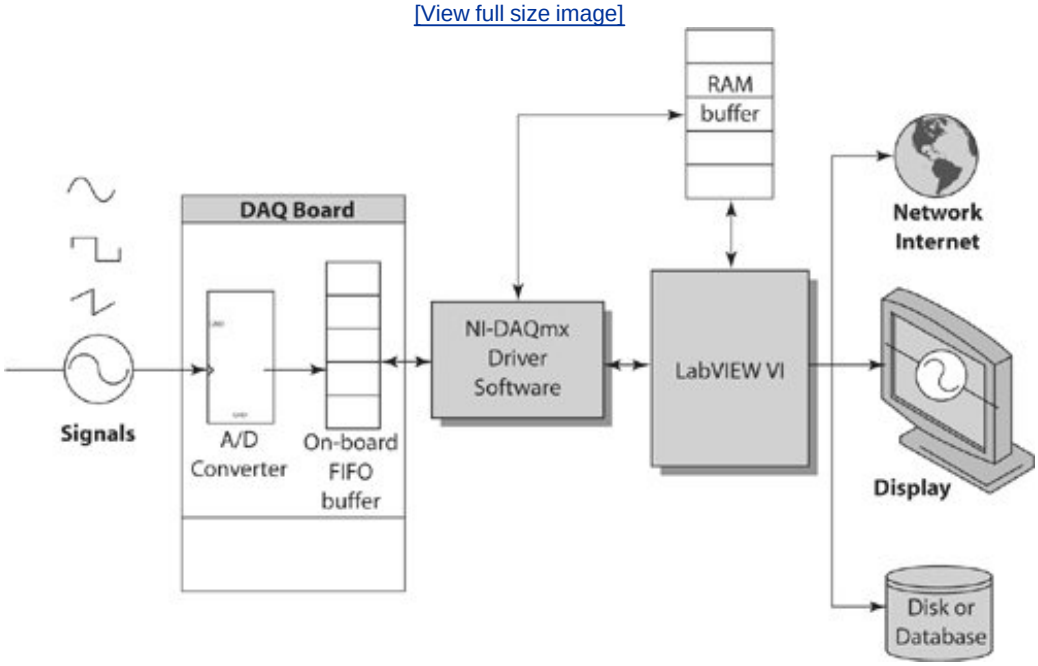
In this section, you will expand on what you learned about analog I/O. You will see how analog data is handled when going from your DAQ device to software, and several considerations for how to acquire data.

The concepts that you are about to learn, buffering and triggering, are also applicable to digital I/O. However, they are much more commonly used for analog I/O, so we will introduce them here.

Figure 11.11 shows the relationship between an analog physical signal and LabVIEW.

Figure 11.11. The path of a signal in a LabVIEW DAQ application

[View full size image]



As you can see in Figure 11.11, initiating a DAQ operation involves LabVIEW calling NI-DAQmx (which is the catch-all driver for the device in use), which in turn signals the hardware to initiate the I/O operation. DAQ devices use on-board *buffers* (called FIFOs, for "First-In, First-Out") and RAM buffers as an intermediate place to store the data they acquire. Also note that the software isn't the only place an initiation of an I/O operation takes placean external piece of hardware can also trigger the operation.

Two important characteristics help classify the type of analog DAQ operation you are performing:

- Whether you use a buffer.
- Whether you use an external trigger to start, stop, or synchronize an operation.

Buffers

A *buffer*, as used in this context, is an area of memory in the PC (not the on-board FIFO) reserved for data to reside in temporarily before it goes somewhere else. For example, you may want to acquire a few thousand data samples in one second. It would be difficult to display or graph all that data in the same one second. But by telling your DAQ device to acquire the data into a memory buffer, you can quickly store the data there first, and then later retrieve it for display or analysis. Remember, buffers

are related to the speed and volume of the DAQ operation (generally analog I/O). If your DAQ device has DMA capability, analog input operations have a faster hardware path to the RAM in your computer, meaning the data can be acquired directly into the computer's memory.

Not using a buffer means you must handle (graph, save to disk, analyze, whatever) each data point *one at a time* as it is acquired because there is no place to "keep" several points of data until you can handle them.

Use buffered I/O when

- You need to acquire or generate many samples at a rate faster than is practical to display, store on a hard drive, or analyze in real time.
- You need to acquire or generate AC data (>10 samples/sec) continuously and be able to do analysis or display of some of the data on the fly.
- The sampling period must be precise and uniform throughout the data samples.

Use nonbuffered I/O when

- The data set is small and short (for example, acquiring one data point from each of two channels every second).
- You need to reduce memory overhead (the buffer takes up memory).

We'll talk more about buffering soon.

Triggering

Triggering refers to any method by which you initiate, terminate, or synchronize a DAQ event. A trigger is usually a digital or analog signal whose condition is analyzed to determine the course of action. In physical instruments such as oscilloscopes, for example, you can similarly use triggers to tell the oscilloscope to wait for a user-specified event such as voltage reaching a certain level or a digital "trigger" input changing state, before drawing a trace signal.

Software triggering is the easiest and most intuitive way to do ityou control the trigger directly from the software, such as using a Boolean front panel control to start/stop data acquisition.

Hardware triggering lets the circuitry in the DAQ device take care of the triggers, adding much more precision and control over the timing of your DAQ events. Hardware triggering can be further subdivided into *external* and *internal* triggering. An example of an internal trigger is to program the DAQ device to output a digital pulse when an analog-in channel reaches a certain voltage level. All National Instruments DAQ devices have an external trigger pin, which is a digital input used for triggering. Many instruments provide a digital output (often called "trigger out") used specifically to trigger some other device or instrument: in this case, the DAQ device.

Use software triggering when

- The user needs to have explicit control over all DAQ operations *and*

- The timing of the event (such as when an analog input operation begins) needn't be very precise.

Use hardware triggering when

- Timing a DAQ event needs to be very precise,
- You want to reduce software overhead (for example, a While Loop that watches for a certain occurrence can be eliminated), *or*
- The DAQ events must be synchronized with an external device.

In the following sections, you will see how you can use the DAQ VIs to configure an I/O operation as buffered or nonbuffered and configure the type of triggering.

Activity 11-2: Buffered Analog Input

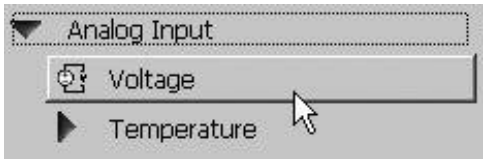
Remember buffered I/O, from the beginning of the chapter? Here is where you get to see it done. Previously, we used a While Loop to continuously acquire multiple points. Now we'll see how we can avoid using a While Loop by programming the hardware to acquire a set number of points at a certain sampling rate.

1. Connect four DC or low-frequency voltage sources to channels 03. If you don't have this many voltage sources, you can connect one source to all channels or make a resistor network to vary the amplitude of the voltage at each channel. Again, if you are using a simulated NI-DAQmx device (as described in Chapter 10), there will be a simulated 10Hz sine wave signal on your simulated device's analog input channelsthese will all be slightly out of phase.

2. Open a new VI, and place a DAQ Assistant VI (Measurement I/O>>NI-DAQmxData Acquisition palette) onto the block diagram.

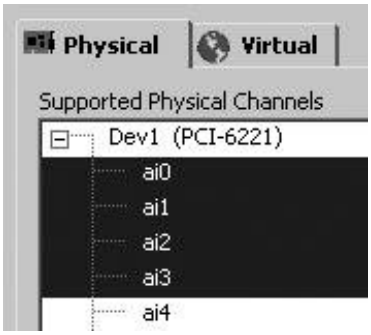
3. From the configuration dialog of your DAQ Assistant VI, choose an Analog Input>>Voltage measurement type (see Figure 11.12).

Figure 11.12. Configuring your task as a voltage analog input type from the DAQ Assistant configuration dialog



4. Select the Physical Channels "ai0" through "ai3" (<Ctrl> or <Shift> click to select multiple channels), and then press the Finish button (see Figure 11.13).

Figure 11.13. Selecting multiple physical channels for your measurement task

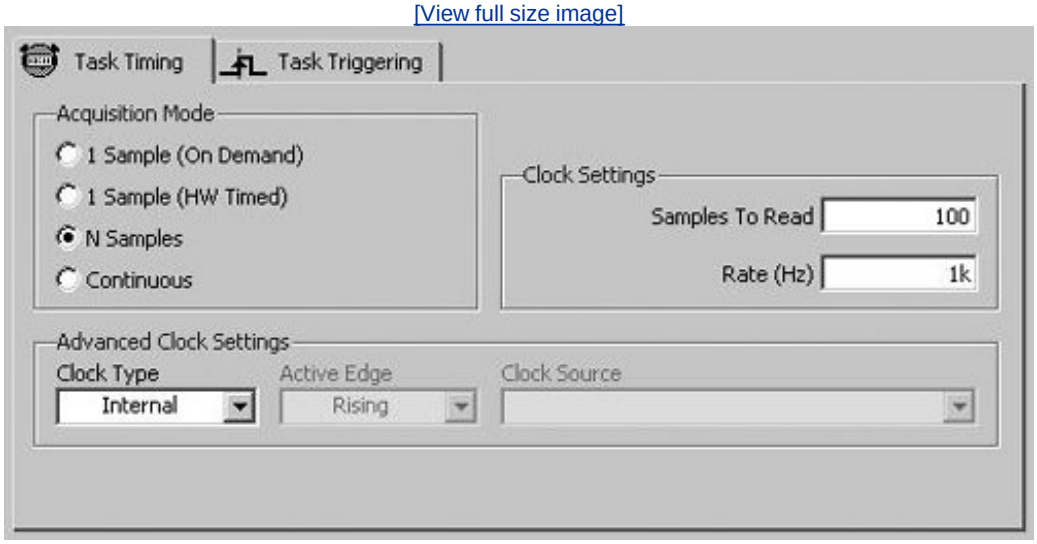


This will complete the process of selecting the measurement type, and you will be presented with the DAQ Assistant task configuration dialog.

5. On the Task Timing tab, select an Acquisition Mode of N Samples. This will configure your task to return multiple measurement values for each channel. On the same tab, select the Samples To Read and Rate (Hz). This will determine how many data points are acquired and the rate they are acquired, each time we call the DAQ Assistant VI (see Figure 11.14).

Figure 11.14. Configuring the task timing acquisition mode and clock settings

[View full size image]



6. Press the OK button to close the DAQ Assistant task configuration dialog. LabVIEW will now build your DAQ Assistant VI.

7. Build the front panel and block diagram shown in Figures 11.15 and 11.16.

Figure 11.15. Front panel of the VI you will create during this activity

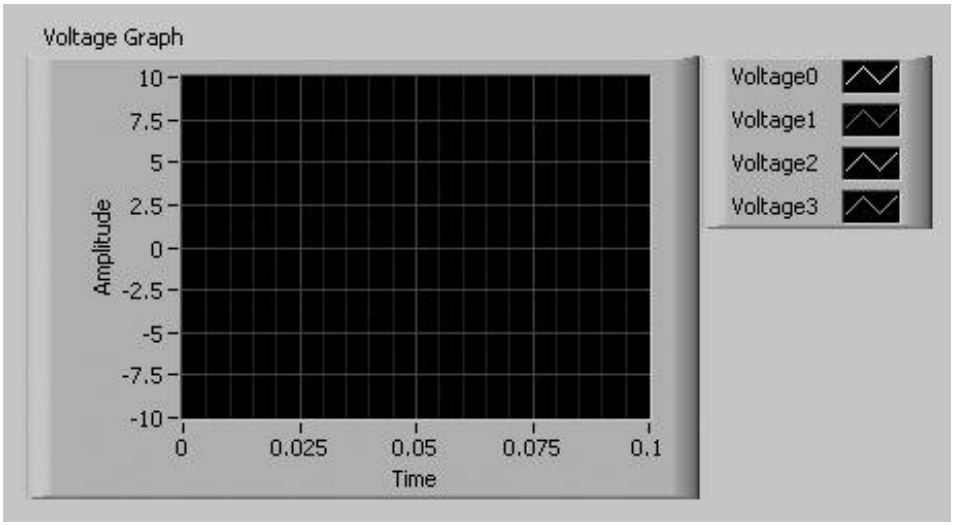


Figure 11.16. Buffered Analog In.vi using DAQ Assistant

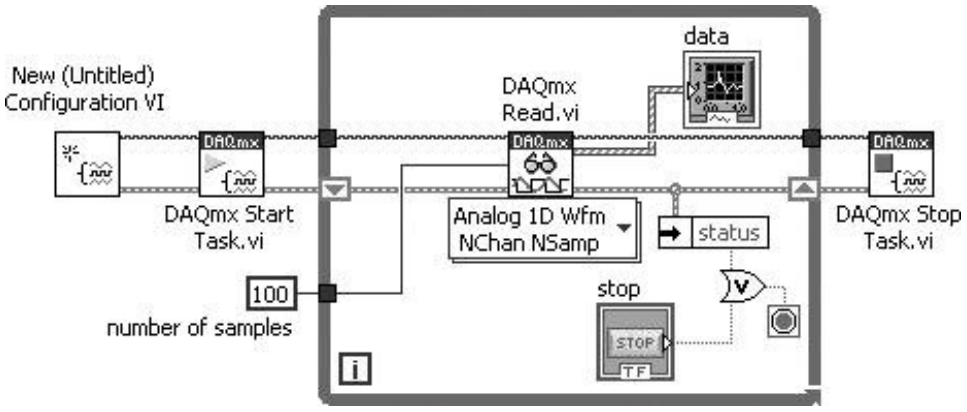
8. Save your VI as `Buffered Analog In.vi`.

9. Run the VI once and look at the graphed data.

10. Let's take a look at what the DAQ Assistant does "behind the scenes." Remember the DAQ Assistant is an Express VI. Like all Express VIs, you can convert the VI to traditional LabVIEW code. Right-click on the DAQ Assistant VI on your block diagram and select "Generate NI-DAQmx Code" from the pop-up menu.

11. After a moment, you will see some LabVIEW code generated on your block diagram. The conversion to code will actually generate a new graph named "data," so delete the "Voltage Graph" you created earlier from the front panel. Then hit <ctrl>-B to delete any leftover broken wires. Your block diagram should look like Figure 11.17. (Annotations have been added, and labels shown, for illustration purposes.)

Figure 11.17. Buffered Analog Inconverted code from DAQ Assistant



12. Notice that the DAQ Assistant has been replaced with some of the VIs from the DAQmx palette. What this code does is actually create and use an NI-DAQmx Task, employing the *Configure*, *Start*, *Read*, and *Stop* NI-DAQmx Task operations.

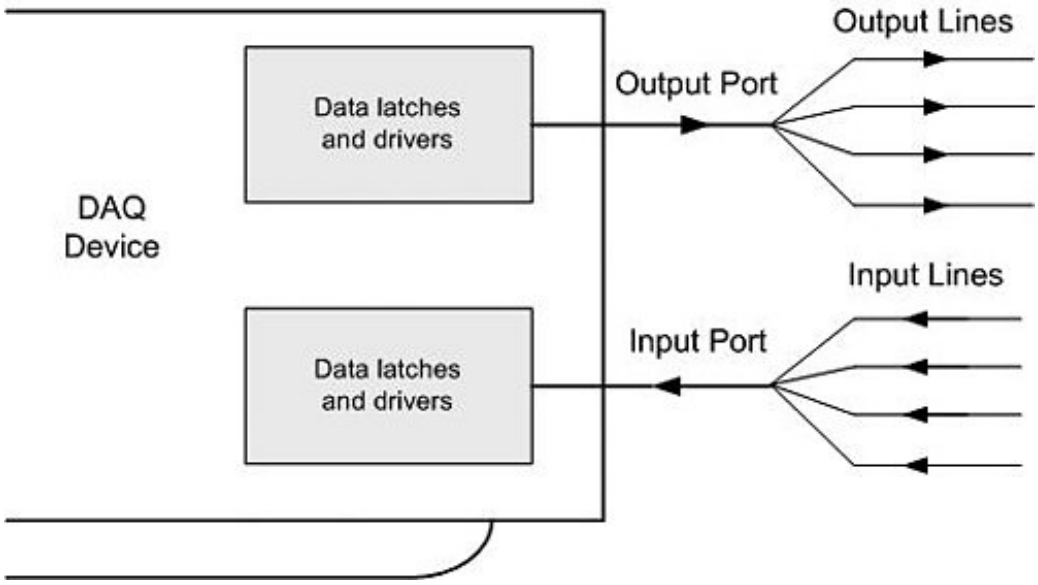
Digital I/O Terms and Definitions

Ah, the digital world, where everythingfor the most partis much simpler than its analog counterpart. 1 or 0, high or low, on or off; that's about it. Actually, some knowledge of binary representation and arithmetic is required, along with knowing the following LabVIEW definitions.

A digital line is the equivalent of an analog channel: a path where a single digital signal is set or retrieved. Digital lines are usually either input lines or output lines, but sometimes can be bi-directional. On most DAQ devices, digital lines must be configured as input or output; they can't act as both at the same time.

A port is a collection of digital lines that are configured in the same direction and can be used at the same time. The number of digital lines in each port depends on the DAQ device, but most ports consist of four or eight lines. For example, a multifunction device could have eight digital lines, configurable as one eight-line port, two four-line ports, or even eight one-line ports. Ports are specified as digital channels, just like analog channels. Figure 11.18 shows the conceptual relationships between digital ports and digital lines on a typical DAQ device.

Figure 11.18. Digital ports and lines on a typical DAQ device



Port width is the number of lines in a port.

State refers to the one of two possible cases for a digital line: a Boolean TRUE (same as logical 1 or "on"), or a Boolean FALSE (same as logical 0 or "off").

A pattern is a sequence of digital states, often expressed as a binary number, which describes the states of each of the lines on a port. For example, a four-line port might be set with the pattern "1101," meaning the first, third, and fourth lines are TRUE, and the second line is FALSE. The first bit, or least-significant bit (LSB) is the rightmost bit on the pattern. The last (fourth in this example), or most-significant bit (MSB) is the leftmost bit in the pattern. This pattern can also be converted from its binary equivalent to the decimal number 13.

Incidentally, National Instruments' DAQ devices use TTL positive logic, which means a logical low is somewhere in the 00.8 V range; a logical high is between 2.2 V and 5.5 V.

If a voltage signal ranging between 0.8 volts and 2 volts were to be sent into the input of a TTL gate, there would be no certain response from the gate. Such a signal is considered uncertain, because it is uncertain how the gate circuit would interpret such a signal.

Activity 11-3: Reading Digital Inputs

Now you will get to read some digital data! You will use the DAQ Assistant Express VI to read digital

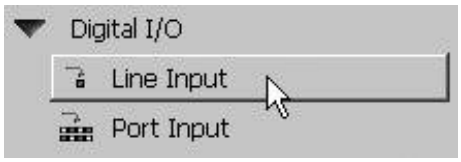
lines on your DAQ device.

1. Connect four digital signals (0 or 5 Volts DC) to port 0, lines 03. If you don't have this many digital signals, you can connect one source to all channels. If you are using a simulated NI-DAQmx device (as described in Chapter 10), there will be a simulated digital signal on your simulated device's digital input channelsthe digital signal will be a binary counter that starts at zero (0) and increments each time you read the data.

2. Open a new VI, and place a DAQ Assistant VI (Measurement I/O>>NI-DAQmxData Acquisition palette) onto the block diagram.

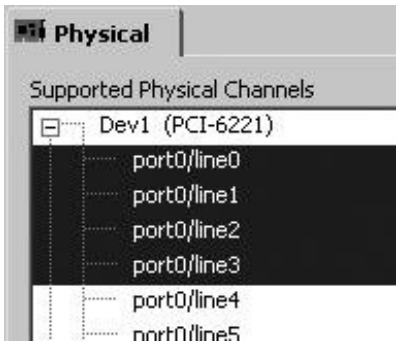
3. From the configuration dialog of your DAQ Assistant VI, choose a Digital I/O>>Line Input measurement type (see Figure 11.19).

Figure 11.19. Configuring your task as a digital line input type from the DAQ Assistant configuration dialog



4. Select your DAQ device and channels "port0/line0" through "port0/line3" (<Ctrl> or <Shift> click to select multiple channels), and then press the Finish button (see Figure 11.20).

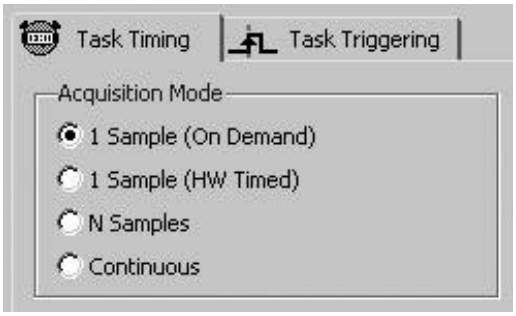
Figure 11.20. Selecting the physicals channel for your measurement task



This will complete the process of selecting the measurement type and you will be presented with the DAQ Assistant task configuration dialog.

5. On the Task Timing tab, select an Acquisition Mode of 1 Sample (On Demand). This will configure your task to return a single measurement value each time you call the DAQ Assistant VI (see Figure 11.21).

Figure 11.21. Configuring the task timing acquisition mode as 1 Sample (On Demand)



6. Press the OK button to close the DAQ Assistant task configuration dialog. LabVIEW will now build your DAQ Assistant VI.

7. Build the front panel and block diagram shown in Figures 11.22 and 11.23.

Figure 11.22. Front panel of the VI you will create during this activity

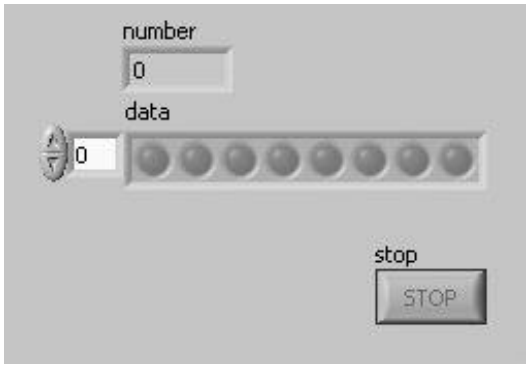
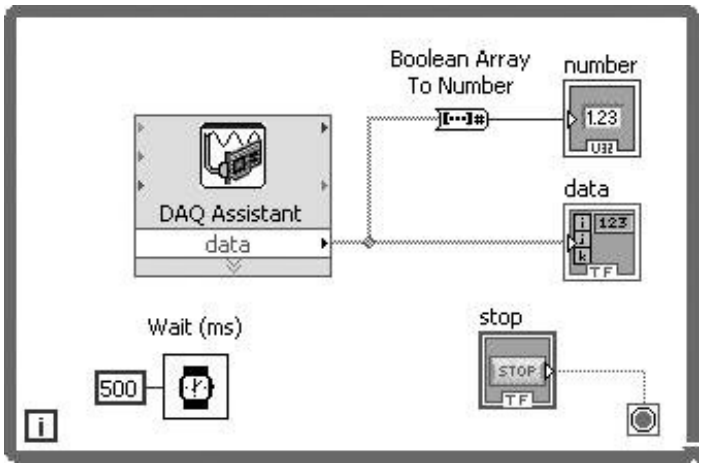


Figure 11.23. Block diagram of the VI you will create during this activity



Boolean Array To Number Function

The Boolean Array To Number function (found on the Programming>> Boolean palette) converts a Boolean array to a 32-bit unsigned integer by interpreting the array as the binary representation of an integer, with the first element of the array being the least significant bit.

8. Save your VI as `Read Digital Inputs.vi`.

9. Run the VI and watch the LEDs change state and the number change values.

Congratulationsyou're reading digital data! If you are feeling adventurous, try using the DAQ Assistant for performing other types of DAQ operations. For example, you could try doing buffered digital acquisition, or continuous digital acquisition. Try configuring a DAQ Assistant VI for writing a digital output and see if you can switch relays or turn your lights on and off. With the DAQ Assistant, it is so easy!

Now we will take a look under the hood and see how NI-DAQmx works, and then start using the DAQmx VIs in LabVIEW to configure our NI-DAQmx tasks programmatically.

PREV NEXT

NI-DAQmx Tasks

In Chapter 10, you learned how to create virtual channels in MAX. These virtual channels correlate with physical channels on your DAQ device that can be read from or written to in LabVIEW using the NI-DAQmx VIs. Now you'll learn about tasks, an even more powerful framework that makes building your DAQ application easier.

Tasks are the key to getting things done in NI-DAQmx. A task is a collection of one or more virtual channels with timing, triggering, and other properties. An NI-DAQmx Task is a neat way to "wrap up" all the parameters related to the data acquisition task.

Let's explore this a little more, because tasks are unlike other structures in LabVIEW and take some getting used to.

Conceptually, a task represents a measurement (input signal acquisition) or generation (output signal generation) you want to perform. All channels in a task must be of the same channel type, such as analog input or counter output. With some devices, you can include channels from multiple devices in a task. To perform a measurement or a generation with a task, you would follow these steps:

1. Create or load a task. You can create tasks interactively with the DAQ Assistant or programmatically in your ADE, or Application Development Environment, such as LabVIEW or LabWindows/CVI.

2. Configure the channel, timing, and triggering properties as necessary.

3. Optionally, perform various task state transitions to prepare the task to perform the specified operation.

4. Read or write samples.

5. Clear the task.

If appropriate for your application, you would repeat steps 2 through 4. For instance, after reading or writing samples, you can reconfigure the virtual channel, timing, or triggering properties, and then read or write additional samples based on this new configuration.

If properties need to be set to values other than their defaults for your task to be successful, your program must set these properties every time it executes. For example, if you run a program that sets property A to a non-default value and follow that with a second program that does not set property A, the second program uses the default value of property A. The only way to avoid setting properties programmatically each time a program runs is to use virtual channels and/or tasks created in the DAQ Assistant.

You've just learned all the basics of how tasks work. Now you will see how to create tasks in MAX that can be used in LabVIEW.

Creating NI-DAQmx Tasks in MAX

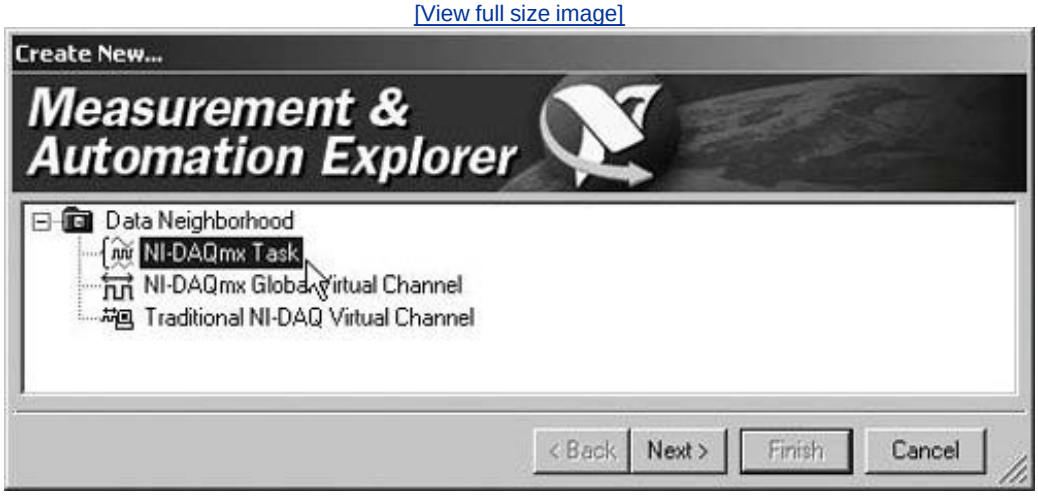
To create an NI-DAQmx Task in MAX, right-click on the Data Neighborhood in the Configuration tree and select Create New . . . from the pop-up menu (see Figure 11.24).

Figure 11.24. Launching the Create New . . . dialog from the Data Neighborhood node in MAX

In the resulting dialog, select NI-DAQmx Task and press the Next button (see Figure 11.25).

Figure 11.25. Selecting NI-DAQmx Task from the Create New . . . dialog

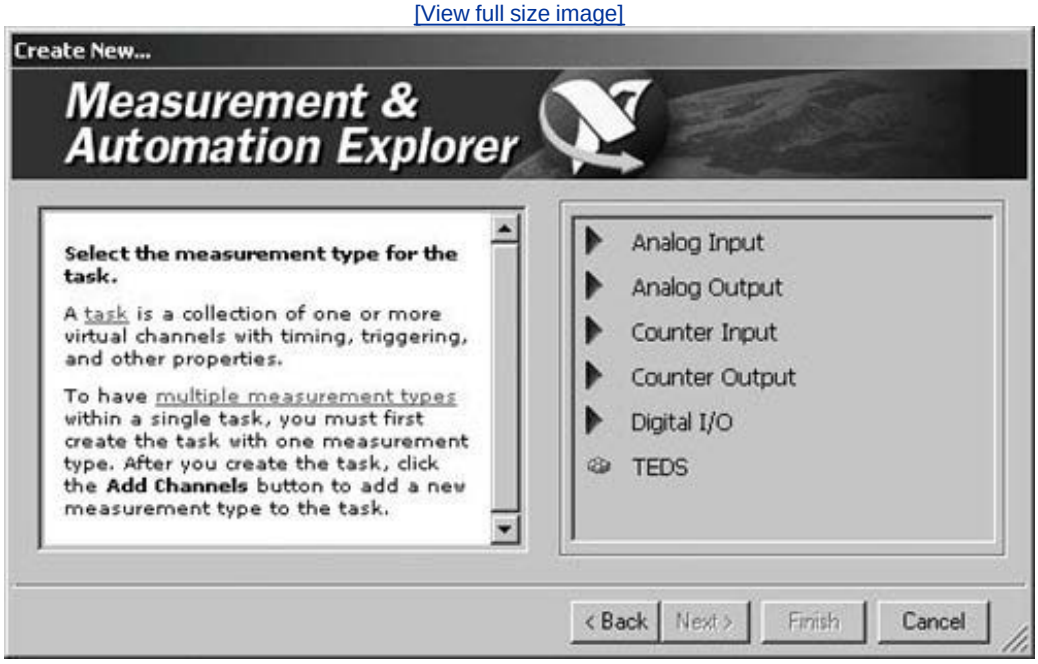
[View full size image]



Select a measurement type (see Figure 11.26).

Figure 11.26. The Create New . . . dialog showing the possible measurement types for your new task

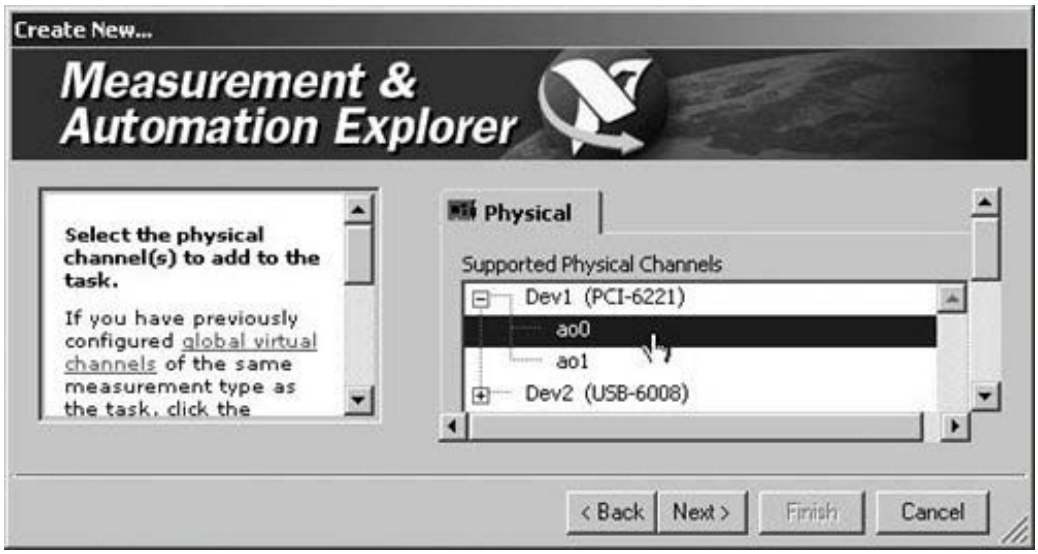
[View full size image]



Select one or more physical channels from the list of devices and physical channels that support the measurement type you have selected, and press the Next button (see Figure 11.27).

Figure 11.27. Selecting one or more physical channels for your measurement task

[View full size image]



Edit the name of the task (or use the default namethis can be changed later) and press the Finish button (see Figure 11.28).

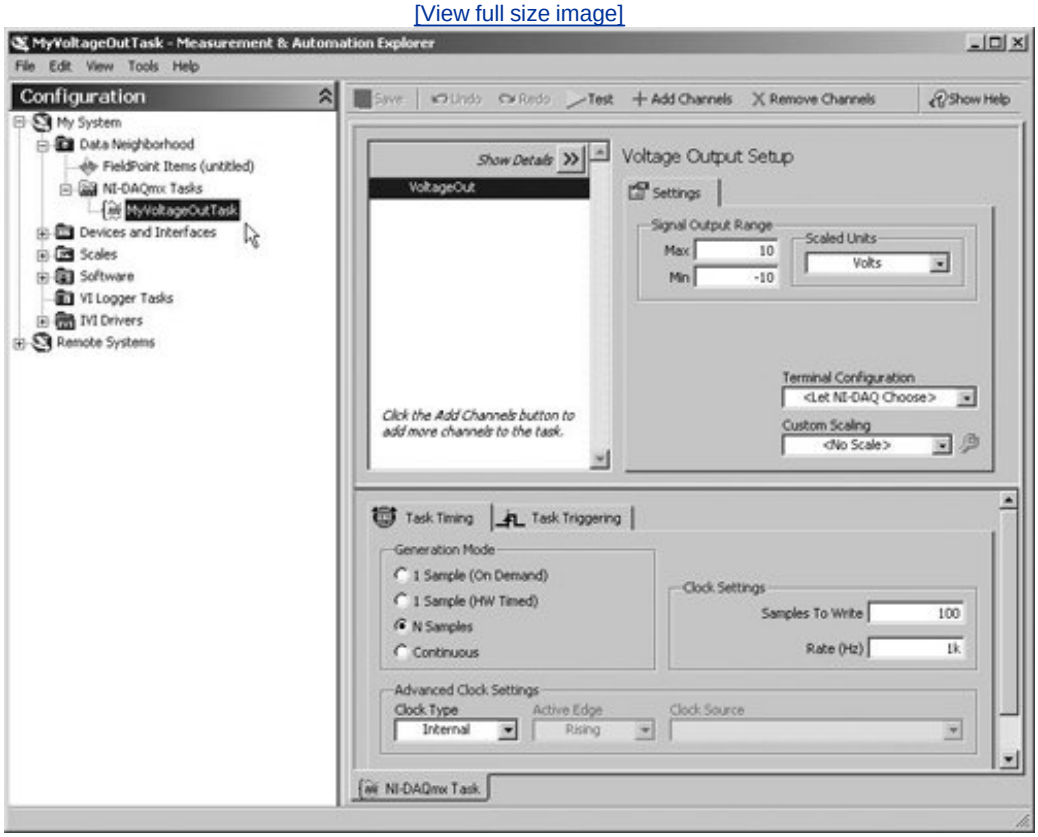
Figure 11.28. Naming your new NI-DAQmx task

Your new task will appear beneath the My System>>Data Neighborhood>> NI-DAQmx Tasks in MAX's Configuration tree (see Figure 11.29). Click on the task with your mouse to view the task's configuration dialog. You can now configure and test the specific functionality of your task. Congratulationsyou're well on your way to using NI-DAQmx tasks in LabVIEW!

Figure 11.29. Viewing your new task's configuration dialog from within

MAX

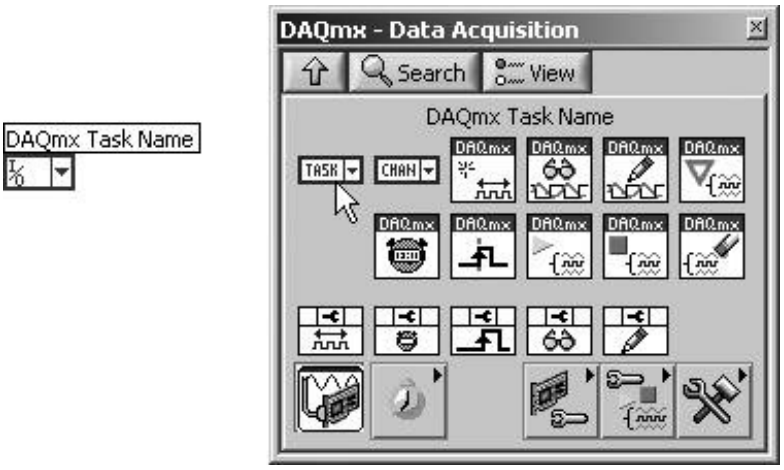
[View full size image]



Referencing MAX DAQmx Tasks in LabVIEW

Now that you know how to create and access NI-DAQmx Tasks in MAX, you will learn how to access them from within LabVIEW. Simply place a DAQmx Task Name constant (found on the Measurement I/O>>DAQmxData Acquisition palette) onto the block diagram, as shown in Figure 11.30.

Figure 11.30. Placing a DAQmx Task Name constant from the palette onto your block diagram

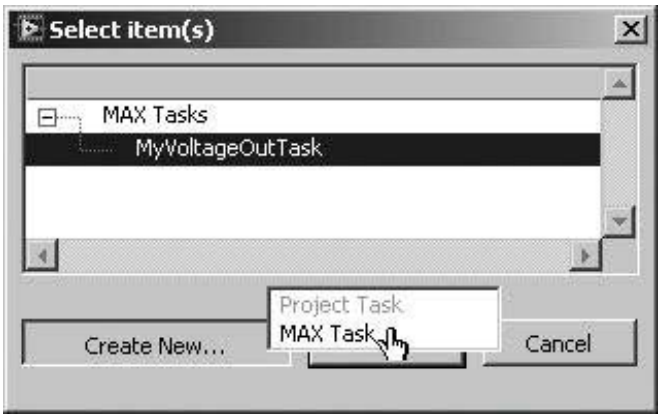


Using the Operating tool, you can select an NI-DAQmx task that you have created in MAX, by clicking on the DAQmx Task Name constant, as shown in Figure 11.31.

Figure 11.31. Selecting an NI-DAQmx task, created in MAX, from the DAQmx Task Name constant's drop-down list

If you would like to create a new NI-DAQmx task in MAX, select the Browse . . . option in the DAQmx Task Name drop-down list, and then choose Create New . . . >>Max Task (see Figure 11.32). This will open the NI-DAQmx task wizardwhich you learned about earlier in the section, "Creating NI-DAQmx Tasks in MAX"where you can create a new NI-DAQmx task, which may then be selected from the DAQmx Task Name constant.

Figure 11.32. Creating a new task from a DAQmx Task Name constant's Browse . . . dialog



Generating Code from MAX DAQmx Tasks

You are now probably wondering how much work you are going to have to do to actually take some data using the NI-DAQmx task that you have created in MAX. As this section's title suggests, LabVIEW will make this very easy for you. Figure 11.33 shows the Generate Code>> options that are available from the DAQmx Task Name constant. Table 11.1 contains descriptions of each of these code generation options. Figure 11.34 shows the Configuration and Example code generated from an analog output NI-DAQmx task.

Figure 11.33. The Generate Code>> pop-up submenu of a DAQmx Task Name constant showing the options for generating LabVIEW code from an NI-DAQmx task

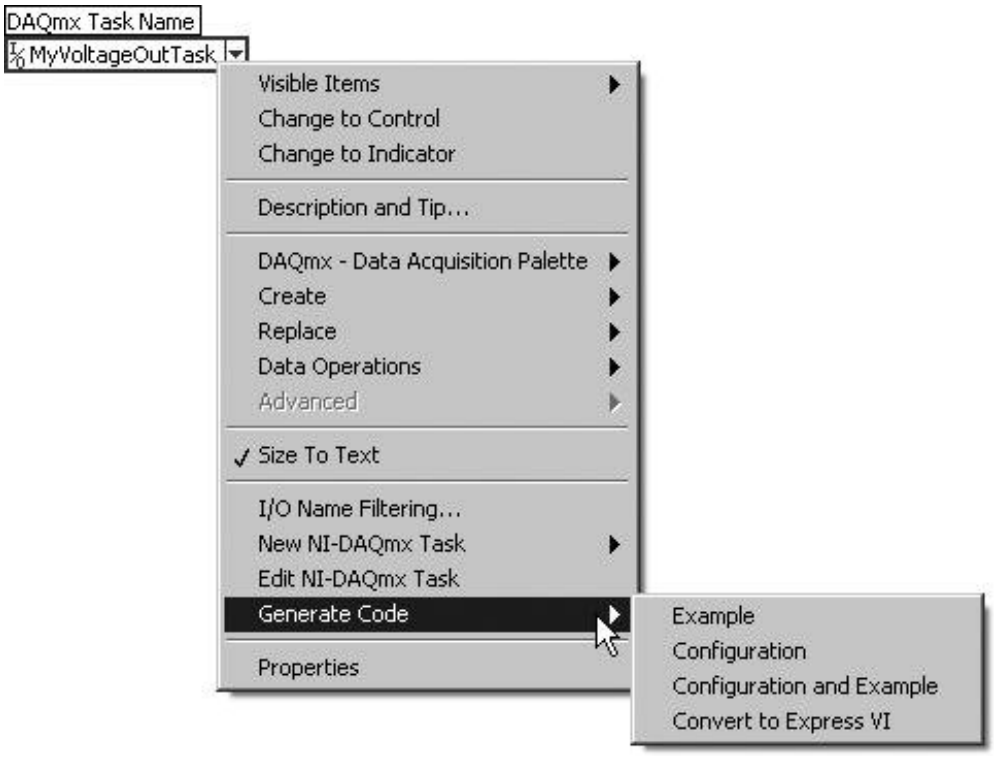
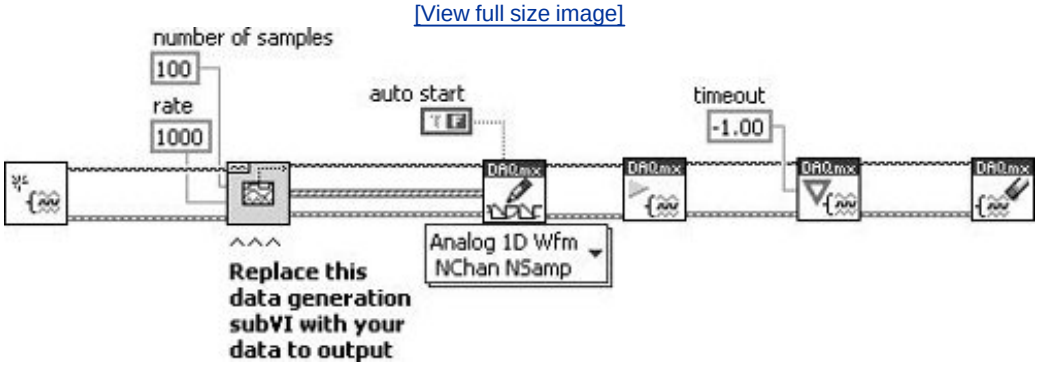


Table 11.1. Code Generation Descriptions

Generate Code	***Description***
Example	Generates all the code necessary to run the task or channel, such as VIs needed to read or write samples, and VIs to start and stop the task, loops, and graphs. Select this option if the task or channel is specific to your system, and you do not use the task or channel on other systems.
Configuration	Generates the code associated with the configuration. LabVIEW replaces the I/O constant or control with a subVI that contains VIs and Property Nodes used for channel creation and configuration, timing configuration, and triggering configuration used in the task or channel. Select this option if you need a portable configuration that you can move to another system.
Configuration and Example	Generates configuration code and example code for the task or channel in one step.
Convert to Express VI	Replaces the DAQmx Task Constant with a DAQ Assistant Express VI. Similarly, you can convert a DAQ Assistant Express VI into a DAQmx Task Constant.

Figure 11.34. Configuration and Example code generated from an analog output NI-DAQmx task

[View full size image]



The wonderful thing about the NI-DAQmx code generation feature is that it teaches you how to use NI-DAQmx by providing you with custom example VIs built just for you! How is that for a personal touch?

Using NI-DAQmx Tasks in LabVIEW

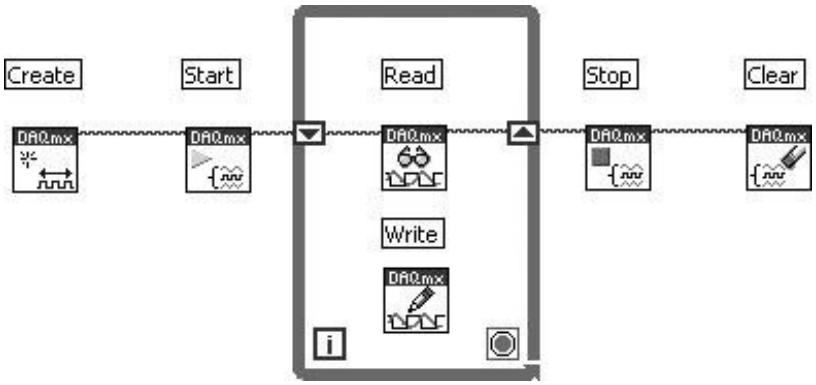
Before we get into the details of how each individual DAQmx VI works, you will want to see a bird's eye view of the DAQmx VIs and methodology. Fortunately, using NI-DAQmx tasks in LabVIEW is very easyit consists of the following steps:

1. Create a task (or reference a MAX DAQmx task).
2. Start the task.
3. Read or Write, and repeat as necessary.
4. Stop the task.
5. Clear the task.

For some measurements and generations, we will need to configure timing, triggering, and other task properties before starting the task. And also, we may need to wait for a task to complete before clearing it.

So, nearly all of your NI-DAQmx applications will be logically organized something like Figure 11.35.

Figure 11.35. Block diagram showing the logical organization of NI-DAQmx applications

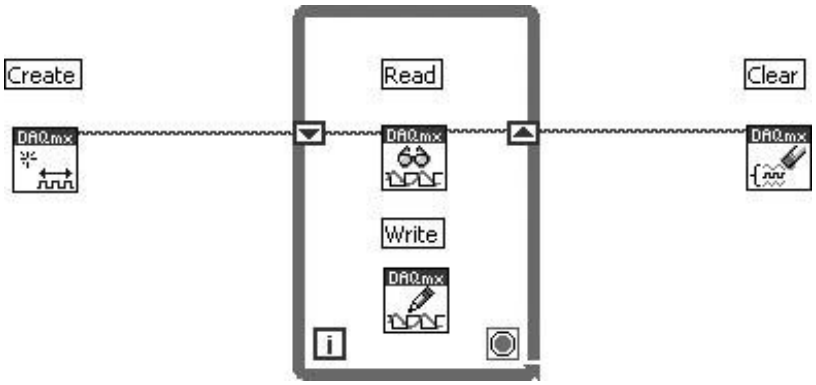


But it is good to know that

- You don't always have to Start your task. The Read or Write operations will usually Start the task automatically.
- You don't have to Stop the task before you Clear it. The Clear operation will Stop the task if it is running.

So, your NI-DAQmx application might look more like Figure 11.36.

Figure 11.36. Block diagram showing the logical organization of NI-DAQmx applications that do not call the (sometimes optional) Start or Stop task operations



NI-DAQmx is starting to look very simple now! In fact, all you have to do, in order to customize the task to your specific type of measurement or generation, is select the appropriate polymorphic VI member for the Create step and the Read or Write step. (We discuss polymorphic VIs at the beginning of Chapter 14, "Advanced LabVIEW Data Concepts." You might want to jump ahead and read that short section and learn how to use the Polymorphic VI Selector. Then come back here and continue to learn about DAQmx.)

We will learn about each of these VIs in the next sections, but we want to show you something very important first.

Many of the DAQmx VIs are polymorphic, and allow you to select a specific instance that matches the task you wish to perform. You make this selection using the Polymorphic VI Selector, shown in Figures 11.37 through 11.40.

Figure 11.37. Create analog input task

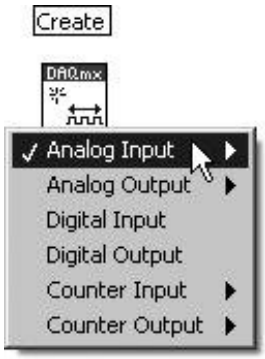


Figure 11.38. Read analog input

Figure 11.39. Create digital output task

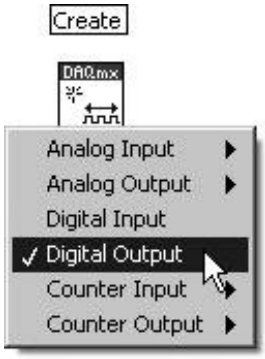


Figure 11.40. Write digital output

The selection made with the Polymorphic VI Selector for the Create step must match the selection for the Read or Write step. If you do not select compatible modes, then the

read or write operation will return an error when you try to run your application.

For example, if you create an *analog input* task, then you can only call the *analog read* operation, as shown in Figures 11.37 and 11.38.

Similarly, if you create a *digital output* task, then you must configure the write function as a *digital write*, as shown in Figures 11.39 and 11.40.

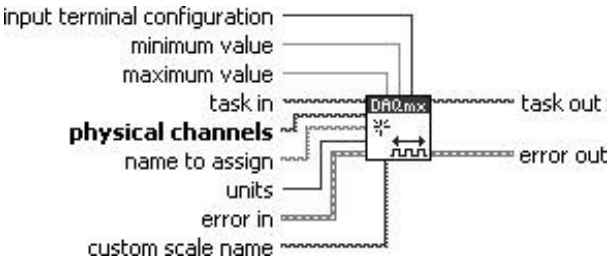
Keep this important fact in mind as you learn the specifics of these DAQmx VIs, next.

DAQmx Create Virtual Channel: Creating Tasks

The first step in using NI-DAQmx in LabVIEW is creating a task. You have already learned how to create a task in MAX and use a DAQmx Task Name control or constant to reference it in LabVIEW. However, you can also create a new task programmatically, using the DAQmx Create Virtual Channel (Measurement I/O>>DAQmx Data Acquisition palette).

DAQmx Create Virtual Channel (Measurement I/O>>DAQmx Data Acquisition palette) creates a virtual channel or set of virtual channels and adds them to a task. The instances of this polymorphic VI correspond to the I/O type of the channel, such as analog input, digital output, or counter output; the measurement or generation to perform, such as temperature measurement, voltage generation, or event counting; and, in some cases, the sensor to use, such as a thermocouple or RTD for temperature measurements.

Figure 11.41. DAQmx Create Virtual Channel



If you use this VI within a loop without specifying a task in, NI-DAQmx creates a new task in each iteration of the loop. Use the DAQmx Clear Task VI within the loop after you are finished with the task to avoid allocating unnecessary memory. Refer to the Task Creation and Destruction sections of the NI-DAQmx online help for more information about when NI-DAQmx creates tasks and when LabVIEW automatically destroys tasks.

The DAQmx Channel properties include additional channel configuration options.

This might be one of the biggest conceptual hurdles that you will face when learning NI-DAQmx: the fact that you use DAQmx Create Virtual Channel to create a new task from one or more physical

channels.

The reason for the confusion is that DAQmx Create Virtual Channel does many things behind the scenes. Here are some important things to know about this VI:

1. If `task in` is left unwired, *a new task will be created*. So, in addition to creating a virtual channel, DAQmx Create Virtual Channel can create a new task. In most cases, this is the preferred method for programmatically creating a new NI-DAQmx task.

2. In addition to creating a new virtual channel from physical channels, the new virtual channel is added to the task. So, we never really interact with the virtual channelwe do not access virtual channel references.

DAQmx Start Task: Running Your Task

Before you can read from or write to a task, you need to start itset it to a *running* state. In some cases, simply calling DAQmx Read or DAQmx Write (which you will learn about shortly) will start the task automatically. However, it is generally best to use DAQmx Start Task.

DAQmx Start Task (Measurement I/O>>DA Qmx Data Acquisition palette) sets the task to the running state to begin the measurement or generation. Using this VI is required for some applications and is optional for others.

Figure 11.42. DAQmx Start Task

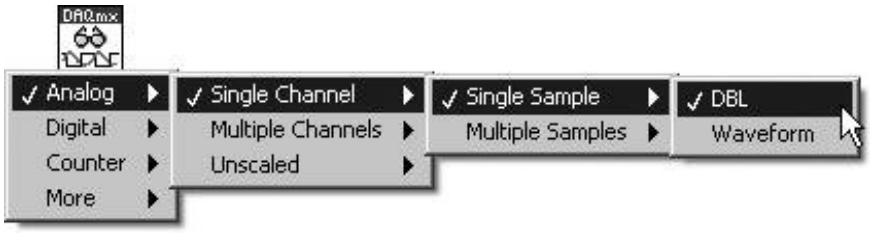
If you do not use this VI, a measurement task starts automatically when the DAQmx Read VI runs. The autostart input of the DAQmx Write VI determines if a generation task starts automatically when the DAQmx Write VI runs.

If you do not use the DAQmx Start Task VI and the DAQmx Stop Task VI when you use the DAQmx Read VI or the DAQmx Write VI multiple times, such as in a loop, the task starts and stops repeatedly. Starting and stopping a task repeatedly reduces the performance of the application.

Read and Write: Measure and Generate

The DAQmx Read and DAQmx Write are polymorphic VIs (which are discussed in Chapter 14) and have over 40 member VIs, each. Use the Polymorphic VI Selector, as shown in Figure 11.43, to select the specific measurement or generation type; but remember that it is critical to choose the correct one that matches the type of measurement or generation task that you have configured.

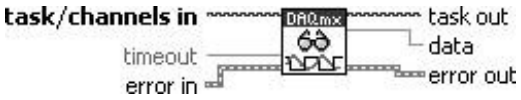
Figure 11.43. The Polymorphic VI Selector menu options of DAQmx Read, used for configuring the VI's mode of operation



Also, it is important to note that the inputs and outputs of VI will change, depending on the measurement or generation type that you choose.

DAQmx Read (Measurement I/O>>DAQmx Data Acquisition palette) reads samples from the task or virtual channels you specify. The instances of this polymorphic VI specify what format of samples to return, whether to read a single sample or multiple samples at once, and whether to read from one or multiple channels.

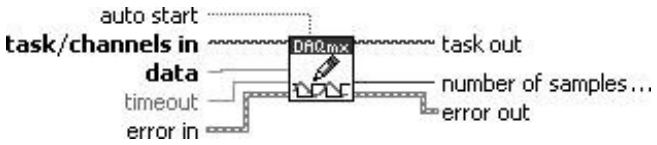
Figure 11.44. DAQmx Read



The DAQmx Read properties include additional configuration options for read operations.

DAQmx Write (Measurement I/O >>DAQmx Data Acquisition palette) writes samples to the task or virtual channels you specify. The instances of this polymorphic VI specify the format of the samples to write, whether to write one or multiple samples, and whether to write to one or multiple channels.

Figure 11.45. DAQmx Write



If the task uses on-demand timing, the default if you do not use the DAQmx Timing VI, this VI returns only after the device generates all samples. If the task uses any timing type other than on-

demand, this VI returns immediately and does not wait for the device to generate all samples. Your application must determine if the task is done to ensure that the device generated all samples.

The DAQmx Write properties include additional configuration options for write operations.

DAQmx Stop Task

After you are done using your measurement or generation task, you will want to stop it (using DAQmx Stop Task)possibly to start it again at a later time.

DAQmx Stop Task (Measurement I/O>>DAQmx Data Acquisition palette) stops the task and returns it to the state the task was in before the DAQmx Start Task VI ran or the DAQmx Write VI ran with the autostart input set to TRUE.

Figure 11.46. DAQmx Stop Task

If you do not use the DAQmx Start Task VI and the DAQmx Stop Task VI when you use the DAQmx Read VI or the DAQmx Write VI multiple times, such as in a loop, the task starts and stops repeatedly. Starting and stopping a task repeatedly reduces the performance of the application.

DAQmx Clear Task

When you are done using your task, use DAQmx Clear Task to release any resources that have been reserved by the task.

DAQmx Clear Task (Measurement I/O>>DAQmx Data Acquisition palette) clears the task. Before clearing, this VI stops the task, if necessary, and releases any resources the task reserved. You cannot use a task after you clear it unless you recreate the task.

Figure 11.47. DAQmx Clear Task

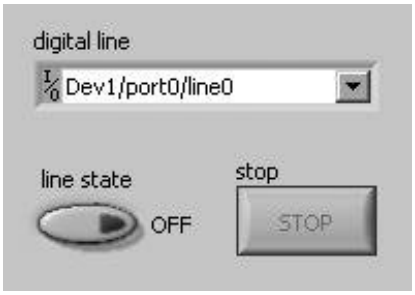
If you use the DAQmx Create Task VI or the DAQmx Create Virtual Channel VI within a loop, use this VI within the loop after you are finished with the task to avoid allocating unnecessary memory.

Activity 11-4: Writing to a Digital Line

You will create a VI that sets the state of a single digital line using DAQmx VIs.

1. Build the front panel shown in Figure 11.48.

Figure 11.48. Front panel of the VI you will create during this activity



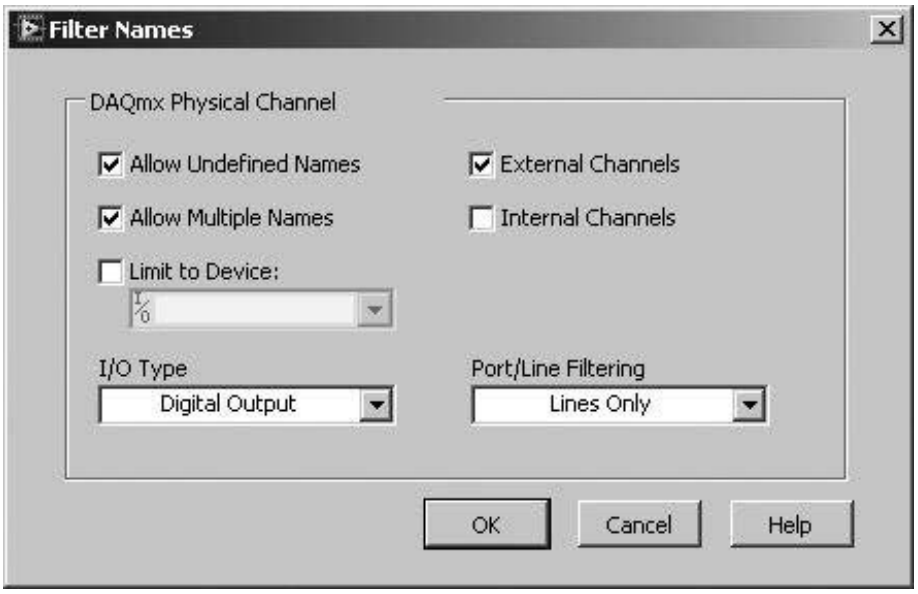
Make sure that the `line state` *Boolean is configured, via its pop-up menu, for Mechanical Action>>Switch When Released. This will ensure that we can set the line state ON and OFF, rather than just ON (which would happen if it "bounced back" automatically after it was pressed).*

DAQmx Physical Channel Control

The `digital line` control on the front panel is a DAQmx Physical Channel (found on the Modern>>I/O>>DAQmx Name Controls palette).

By default, the DAQmx Physical Channel control will only display analog input channels. Pop up on it and select I/O Name Filtering to open the Filter Names dialog (see Figure 11.49). Set the I/O Type to Digital Output and the Port/Line Filtering to Lines Only. This will cause the drop-down list of physical channels to be restricted to digital output lines.

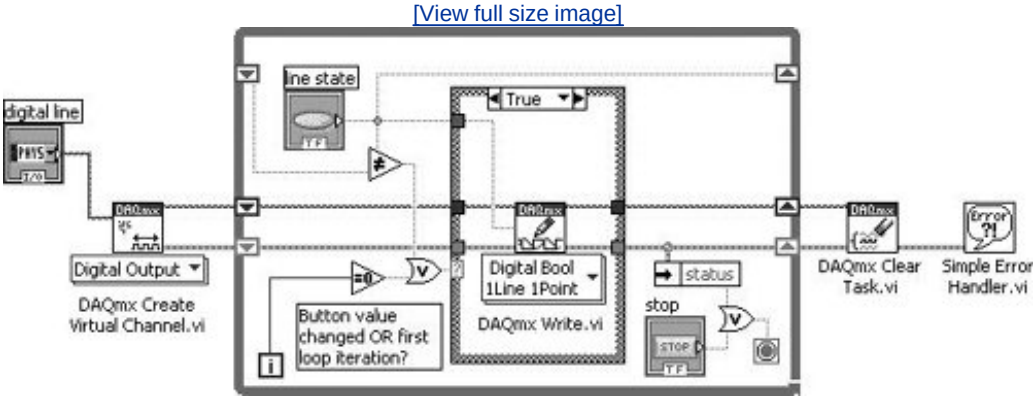
Figure 11.49. Filter Names dialog for configuring DAQmx Physical Channel controls, indicators, and constants



2. Build the block diagram shown in Figure 11.50. (Note that we write the `line state value to the digital line`, whenever the button value has changed OR when the loop count is zero.) All DAQmx VIs used in this activity may be found on the Measurement I/O>>DAQmx Data Acquisition palette.

Figure 11.50. Block diagram of the VI you will create during this activity

[View full size image]



3. Configure DAQmx Create Virtual to create a digital output task by setting the Polymorphic VI Selector to Digital Output.

4. Configure DAQmx Write to write one value to a single digital line by setting its Polymorphic VI Selector to Digital>>Single Channel>>Single Sample>> Boolean (1 line).

DAQmx Write

5. Save the VI as `Write to Digital Line.vi` in your `MYWORK` folder.

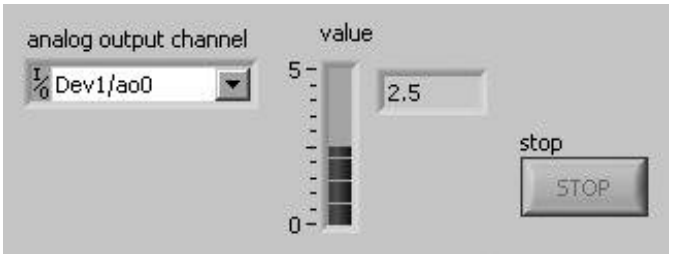
Configure the `digital line` on the front panel and run your VI. Toggle the state and read the voltage on the digital output line. Watch it change between 0 VDC and +5 VDC.

Activity 11-5: Writing a Continuous Analog Waveform

You will create a VI that writes a continuous analog waveform to an analog output. You will write one point at a time, iteratively, in a While Loop.

1. Build the front panel shown in Figure 11.51.

Figure 11.51. Front panel of the VI you will create during this activity

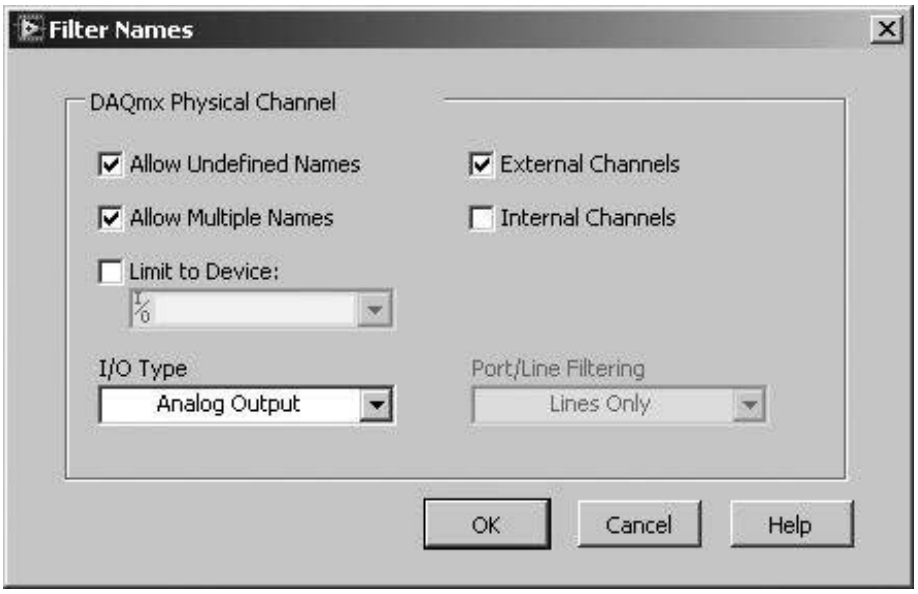


DAQmx Physical Channel Control

The `analog output channel` control on the front panel is a DAQmx Physical Channel (found on the Modern>>I/O>>DAQmx Name Controls palette).

By default, the DAQmx Physical Channel control will only display analog input channels. Pop up on it and select I/O Name Filtering to open the Filter Names dialog. Set the I/O Type to Analog Output. This will cause the drop-down list of physical channels to be restricted to analog output lines (see Figure 11.52).

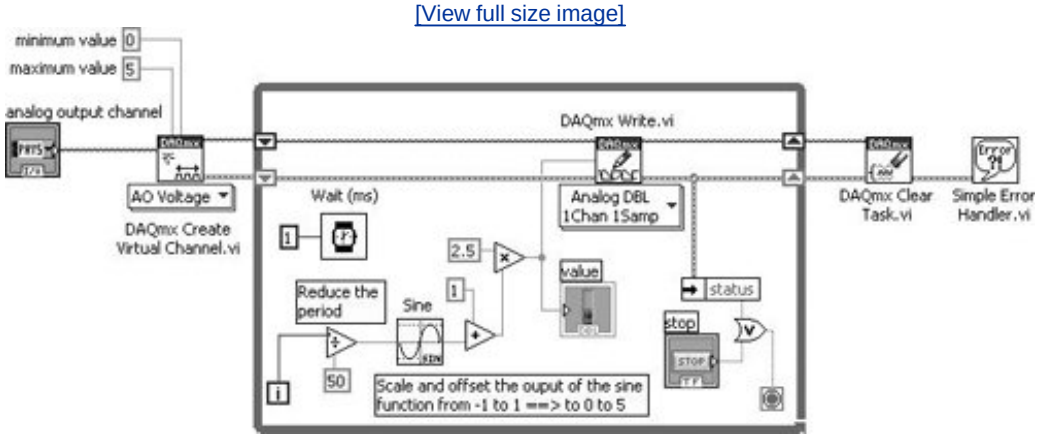
Figure 11.52. Filter Names dialog showing the settings you will need for your DAQmx Physical Channel control in this activity



2. Build the block diagram shown in Figure 11.53. All DAQmx VIs used in this activity may be found on the Measurement I/O>>DAQmx Data Acquisition palette.

Figure 11.53. Block diagram of the VI you will create during this activity

[View full size image]



3. Configure DAQmx Create Virtual to create an analog voltage output task by setting the Polymorphic VI Selector to Analog Output>>Voltage. Set its minimum and maximum values to 0 and 5 Volts.

4. Configure DAQmx Write to write one value to a single analog output line by setting its Polymorphic VI Selector to Analog>>Single Channel>>Single Sample>>DBL.

5. Save the VI as `Write Continuous Analog Waveform.vi` in your `MYWORK` folder.

Configure the `analog output channel` on the front panel and run your VI. Measure the voltage on the analog output line. Watch the sinusoid signal cycle between 0 VDC and +5 VDC.

PREV NEXT

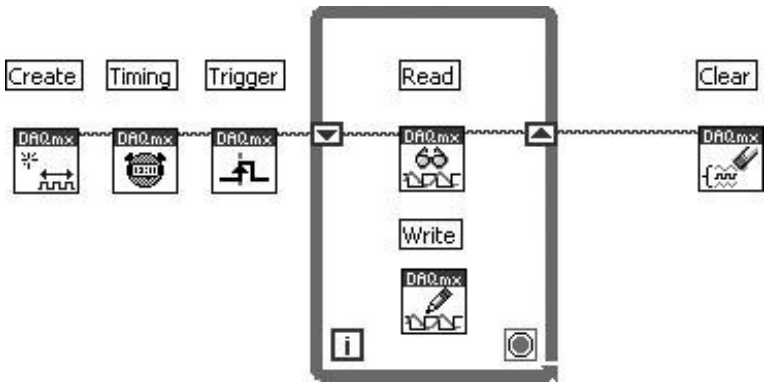
Advanced Data Acquisition

In this section, we'll look at some more advanced scenarios in data acquisition, such as timing and triggering, continuous data acquisition, streaming data to disk, and counting events.

DAQmx Timing and DAQmx Trigger

As we mentioned at the beginning of this section, you can configure timing and triggering options before you start using your task. In this case, your NI-DAQmx applications will be logically organized something like Figure 11.54.

Figure 11.54. Block diagram showing the logical organization of a typical NI-DAQmx application



Both DAQmx Timing and DAQmx Trigger are polymorphic VIs. After placing them on the block diagram, use the Polymorphic VI Selector ring to choose a specific instance.

Figure 11.55. DAQmx Timing

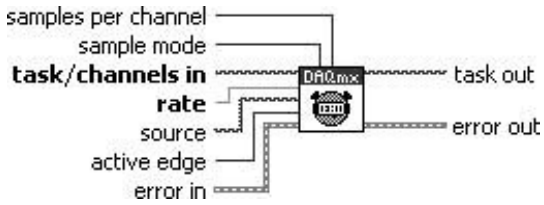
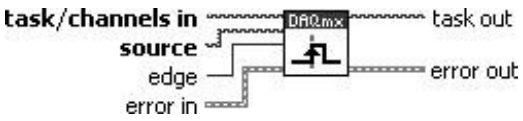


Figure 11.56. DAQmx Trigger



DAQmx Timing (Measurement I/O>> DAQmx Data Acquisition palette) configures the number of samples to acquire or generate and creates a buffer when needed. The instances of this polymorphic VI correspond to the type of timing to use for the task. The DAQmx Timing properties include all timing options included in this VI and additional timing options.

DAQmx Trigger (Measurement I/O>>DAQmx Data Acquisition palette) configures triggering for the task. The instances of this polymorphic VI correspond to the trigger and trigger type to configure.

The DAQmx Trigger properties include all triggering options included in this VI and additional triggering options.

Now, let's take what you've learned about NI-DAQmx timing and triggering and build a VI using both of these features.

Activity 11-6: Triggered Data Acquisition Using Tasks

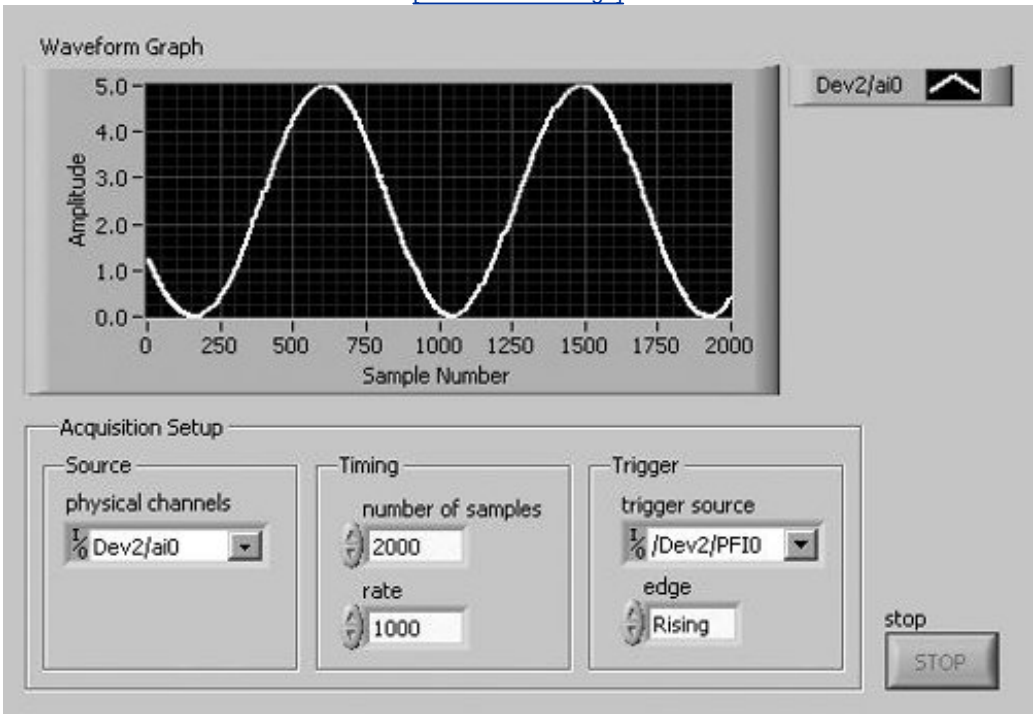
You will now learn how to use the DAQmx Timing and DAQmx Triggering VIs to acquire a buffered analog input signal when triggered by a digital line. Each time the trigger event occurs, you will read the acquired data and display it in a waveform graph. You will use the VI created in Activity 11-4 to generate the trigger signal needed for acquiring the data and the VI created in Activity 11-5 to generate the analog signal that will be acquired.

For this activity to run, you will need a physical NI-DAQmx device. The signal connection (wiring) table is at the end of this activity.

1. Build the front panel shown in Figure 11.57.

Figure 11.57. Front panel of the VI you will create during this activity

[View full size image]



DAQmx Physical Channel Control

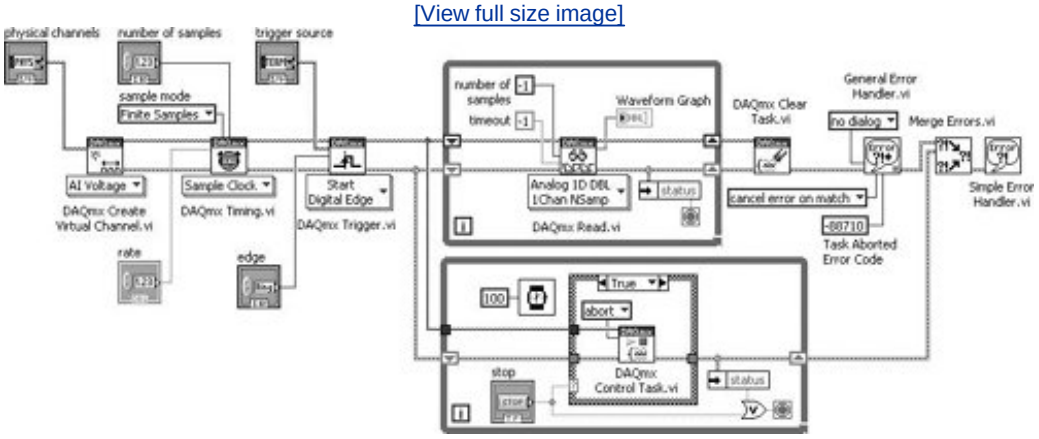
The `physical channels` control on the front panel is a DAQmx Physical Channel (found on the Modern>>I/O>>DAQmx Name Controls palette).

2. Build the block diagram shown in Figure 11.58. All DAQmx VIs used in this activity are found on the Measurement I/O>>DAQmx Data Acquisition palette, with the exception of DAQmx Control Task, which is found on the Measurement I/O>>DAQmx Data Acquisition>>DAQmx Advanced Task Options palette.

DAQmx Control Task

Figure 11.58. Block diagram of the VI you will create during this activity

[View full size image]



3. Configure DAQmx Create Virtual to create an analog input voltage task by setting the Polymorphic VI Selector to Analog Input>>Voltage.

4. Configure the task for finite acquisition by first setting the Polymorphic VI Selector of DAQmx Timing to Sample Clock (Analog/Counter/Digital) and then setting the sample mode input to Finite Samples. You can create the sample mode ring constant by right-clicking on the sample mode input and selecting Create>>Constant from the pop-up menu.

5. Configure the task for Digital Edge start triggering by setting the Polymorphic VI Selector of DAQmx Trigger to Start>>Digital Edge.

There are a lot of useful triggering optionsdigital edge triggering is just one possibility. You can also trigger off a digital pattern (the state of multiple lines in a port), an analog edge (when an analog signal crosses some threshold), or an analog window (only acquire data when an analog signal is within a specified range). This is a very powerful feature of NI-DAQmx!

6. Configure DAQmx Read to return a 1D DBL Array from a single channel by setting its Polymorphic VI Selector to Analog>>Single Channel>>Multiple Samples>>1D DBL.

DAQmx Read

7. Allow the acquisition to be aborted by placing DAQmx Control Task (found on the Measurement I/O>>DAQmx Data Acquisition>>DAQmx Advanced Task Options palette) in the lower loop, and setting its action input to abort. You can create the action ring

constant by right-clicking on the action input and selecting Create>>Constant from the pop-up menu.

DAQmx Control Task

Sometimes errors are good! For example, in this activity, the upper loop will keep running until an error occurs (note that the error cluster is wired to the Conditional Terminal of the upper loop, which is set to Stop if True). We will intentionally generate an error that occurs in the upper loop by calling DAQmx Control Task (in the lower loop) with the abort action. When the DAQmx Read is aborted, it will generate error code -88710, which basically means "the read operation failed, because the task was aborted." Because error -88710 is expected (and intentional, as our way of exiting the loop), we do not want to display this error to the end user. So, the General Error handler is used to cancel that specific error (and only that error).

8. Use the General Error Handler (found on the Programming>>Dialog and User Interface palette) to filter error code -88710, by setting type of dialog to no dialog, exception action to no action, and exception code to `-88710`.

General Error Handler

9. Use Merge Errors (found on the Programming>>Dialog and User Interface palette) to combine the error clusters of the upper and lower loops. This will ensure that if an error occurs in either loop, we will catch it.

Merge Errors

10. Save the VI as `triggered Buffered Analog In.vi` in your `MYWORK` folder.

11. Before you run your VI, you will need to connect your signals. Use the signal connection list in Table 11.2 to help you.

Table 11.2. Signal Connections

Description	*Input*	*Output*
Connect analog output channel 0 to analog input channel 0. Connect reference channel to ground.	dev1/ai0 dev1/ai4	dev1/ao0 dev1/gnd
Connect the port 0 line 0 digital output to the programmable function input (PFI) trigger line 0.	dev1/pfi0	dev1/port0/line0

12. Open Write to Digital Line.vi that you created in Activity 11-4. You will use this VI to generate the signal that triggers the analog read task.

13. Open Write Continuous Analog Waveform.vi that you created in Activity 11-5. You will use this VI to generate the analog signal that will be acquired.

14. Configure the acquisition setup parameters on the front panel (Source, Timing, and Trigger) and run your VI. Toggle the `digital output` Boolean in Write to Digital Line.vi to TRUE.

Congratulations. You've just created some very powerful data acquisition tools! You will be able to use these VIs, and variations of them, to achieve all sorts of functionalityyour imagination is the limit!

Multichannel Acquisition

Earlier, you did an activity where you acquired several channels at once. This is known as multi-channel acquisition.

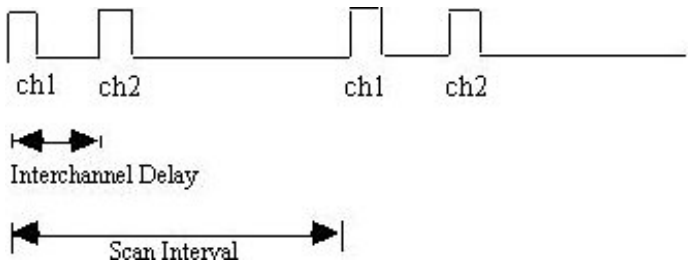
This is a good time to point out the distinction between scan rate and sampling rate:

- Sampling rate refers to how often a measurement should be taken for a particular data channel.
- Scan rate refers to how often a data acquisition board samples ALL the channels, usually in quick successive fashion (hence the term "scan").

On DAQ boards, we usually talk about setting the scan rate. Most of the time scan rate = sampling rate. However, that is not always true, and there is an important limitation of multi-channel I/O that you need to be aware of.

If you set a high scan rate for multiple channels, and observe the data from each channel over actual time (not over array index values), you will notice a successive *phase delay* between each channel. Why is this? Most DAQ devices can only perform one A/D conversion at a time. That's why it's called scanning: The data at the input channels is digitized sequentially one channel at a time. A delay, called the *interchannel delay* (depicted in Figure 11.59), arises between each channel sample.

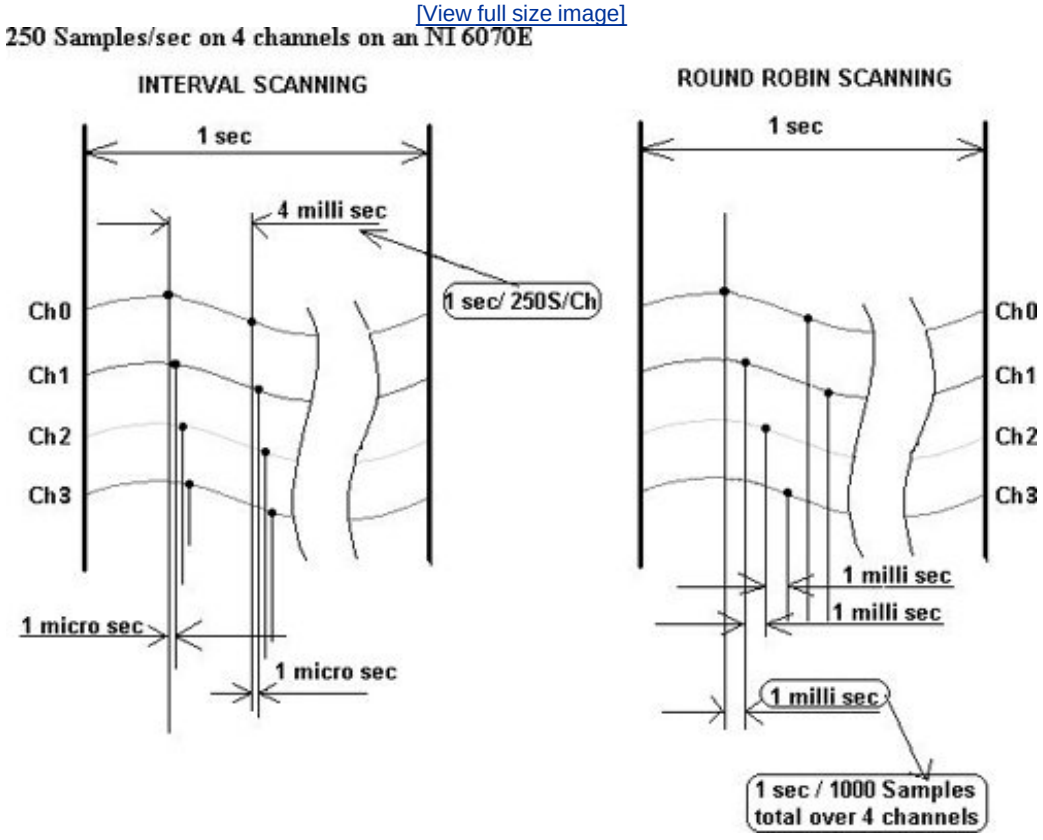
Figure 11.59. Interchannel delay



By default, the interchannel delay is as small as possible, but is highly dependent on the DAQ device. A few DAQ devices, such as the National Instruments S Series devices, do support simultaneous sampling of channels. Other devices support either *interval scanning* or *round robin scanning* (depicted in Figure 11.60). Devices that support interval scanning have a much smaller interchannel delay than those that use round robin scanning.

Figure 11.60. Interval scanning (left) and round robin scanning (right)

[View full size image]



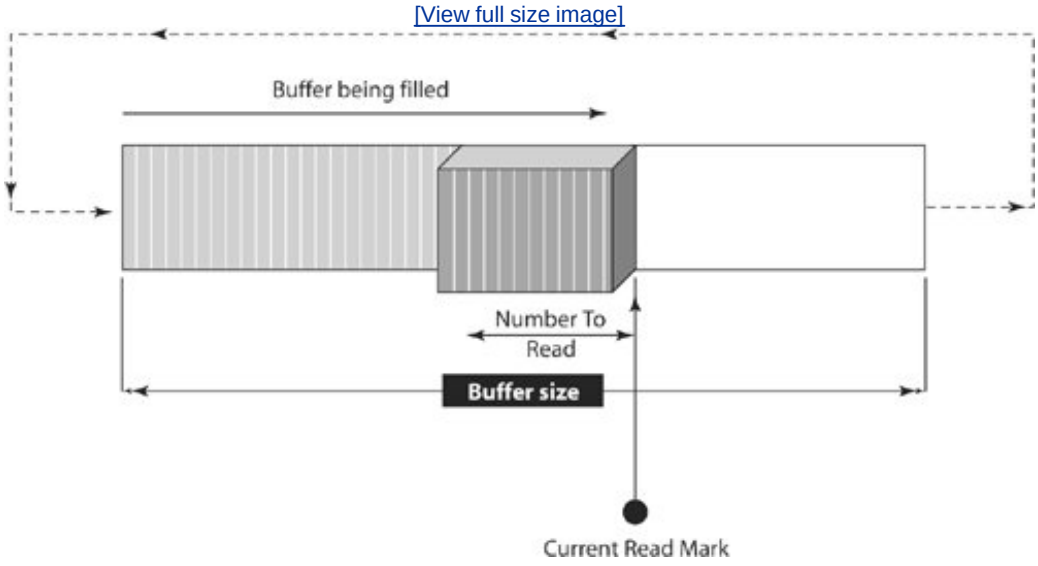
With DC and low-frequency signals, this phase delay in scans is generally not a problem. The interchannel delay may be so much smaller than the scan period that the DAQ device *appears* to virtually sample each channel simultaneously: for example, the interchannel delay may be in the microsecond range and the sampling rate may be 1 scan/sec. However, at higher frequencies, the delay can be very noticeable and may present measurement problems if you are depending on the signals being synchronized.

Continuous Data Acquisition

Continuous data acquisition, or real-time data acquisition, returns data from an acquisition in progress without interrupting the acquisition. This approach usually involves a circular buffer scheme, as shown in Figure 11.61. You specify the size of a large circular buffer. The DAQ device collects data and stores the data in this buffer. When the buffer is full, the DAQ device starts writing data at the beginning of the buffer (writing over the previously stored data, whether or not it has been read by LabVIEW). This process continues until the system acquires the specified number of samples, LabVIEW clears the operation, or an error occurs. Continuous data acquisition is useful for applications such as streaming data to disk and displaying data in real time.

Figure 11.61. Circular buffer

[View full size image]



In the next activity, you will create a VI that acquires data continuously and plots the most recent data.

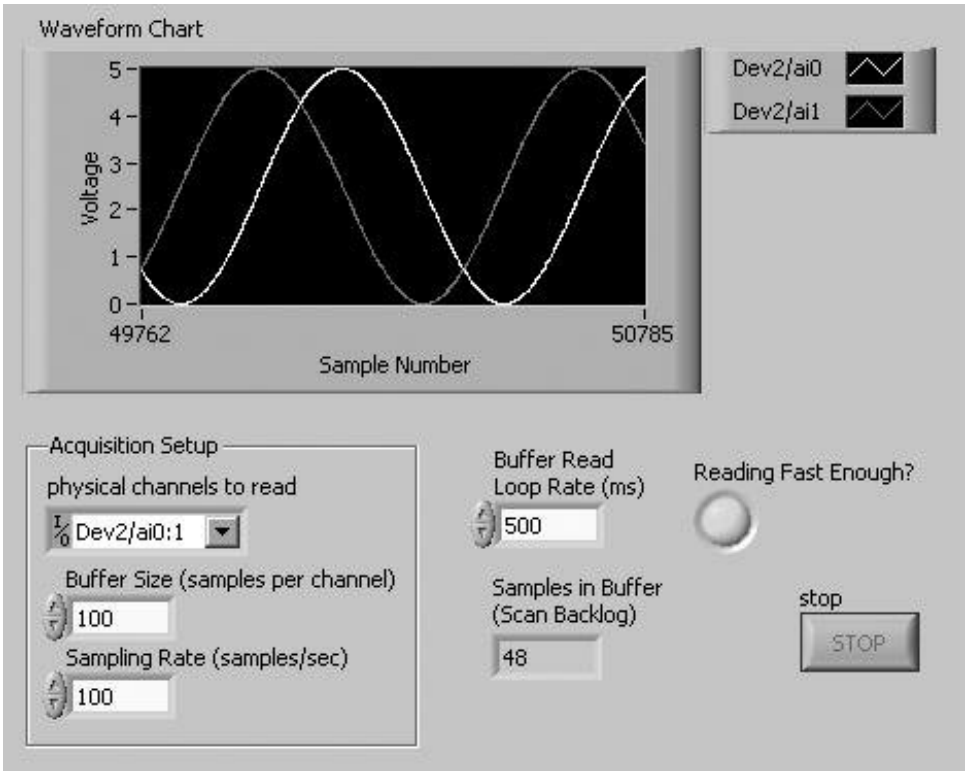
Activity 11-7: Continuous Acquisition

You will create a VI that performs a continuous acquisition by configuring NI-DAQmx to acquire data

into a circular buffer. You will read data out of the circular buffer as it becomes available and display it on a waveform chart. You will also validate that you are reading data fast enough (and that the buffer has not overflowed, causing data loss) by checking whether the amount of data available is less than the size of the circular buffer.

1. Build the front panel shown in Figure 11.62.

Figure 11.62. Front panel of the VI you will create during this activity



Make sure to use a Waveform Chart control (rather than a Waveform Graph) because we want the chart to display all the data that we have read, not just the data that we read each time we read data from the circular buffer.

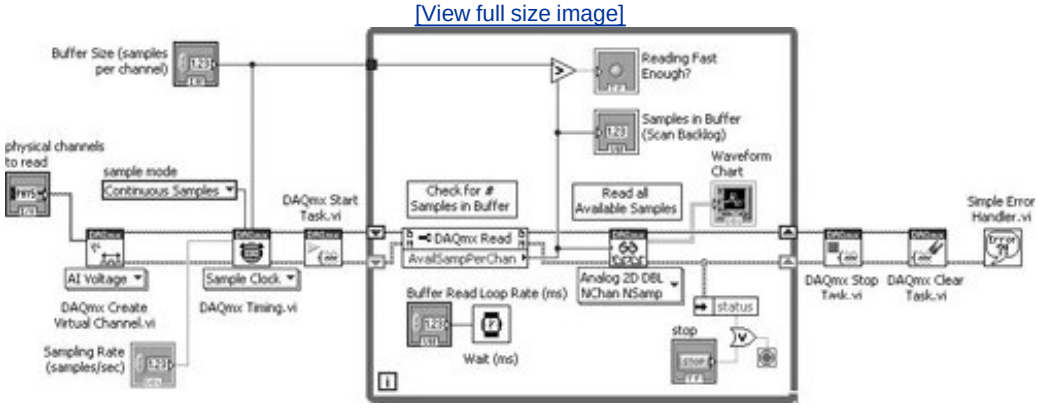
DAQmx Physical Channel Control

The `physical channels to read` control on the front panel is a DAQmx Physical Channel (found on the Modern>>I/O>>DAQmx Name Controls palette).

2. Build the block diagram shown in Figure 11.63. All DAQmx VIs used in this activity may be found on the Measurement I/O>>DAQmx Data Acquisition palette.

Figure 11.63. Block diagram of the VI you will create during this activity

[View full size image]



3. Configure DAQmx Create Virtual to create an analog input voltage task by setting the Polymorphic VI Selector to Analog Input>>Voltage.

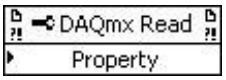
4. Configure the DAQmx Timing to set the task to continuous acquisition mode by setting the Polymorphic VI Selector to Sample Clock (Analog/Counter/Digital) and setting the sample mode input to Continuous Samples.

5. Configure the DAQmx Read Property Node to read the Status>>Available Samples Per Channel attribute. This property tells you the number of samples that are in the circular buffer. You will pass this value into DAQmx Read, in order to read all of the available data in the buffer.

DAQmx Read Property Node

DAQmx Read Property Node (Measurement I/O>>DAQmx Data Acquisition palette) is a Property Node with the DAQmx Read class preselected (see Figure 11.64). Right-click the Property Node and choose Select Filter from the shortcut menu to make the Property Node show only the properties supported by a particular device installed in the system or supported by all the devices installed in the system.

Figure 11.64. DAQmx Read Property Node



6. Configure DAQmx Read to read the data as a 2D array (channels are in rows and samples are in columns) by setting the Polymorphic VI Selector to Analog>>Multiple Channels>>Multiple Samples>>2D DBL.

7. Save the VI as `Continuous Acquisition.vi` in your `MYWORK` folder.

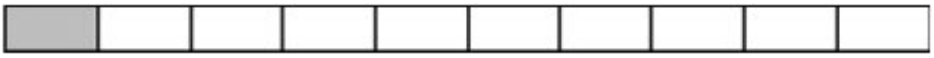
Now configure the acquisition setup parameters on the front panel and run your VI. Note that your loop rate must be fast enough that the buffer does not overflow! Adjust the Buffer Read Loop Rate so that each loop cycle time is less than the time it takes to fill the buffer. You can calculate the time that it takes to fill the buffer using the following, simple equation:

```
Time to Fill the Buffer = Buffer Size / Sampling Rate
```

The Continuous Acquisition VI that you just created is designed to run continuously, or "forever." In order to accomplish this without running out of memory, a fixed-length buffer is allocated, filled up with samples of data from beginning to end, and then the data at the beginning is overwritten as the buffer is filled up again. As an example, suppose this VI were run with a buffer size of 10 and the sampling rate was once per second. Here are snapshots of how the buffer is being filled up with data:

Before the VI is run, the buffer is empty:

After one second, one sample is filled:



After nine seconds, the buffer is almost full:

After twelve seconds, the buffer has filled up and the data at the beginning is being overwritten:

Why the `Samples in Buffer (Scan Backlog)` indicator? It's pretty useful to know if LabVIEW is keeping up with reading the data. If the buffer fills up faster than your VI can retrieve the data in it, you will start losing some of this data, because the buffer will be overwritten.

Are you ready for some TLAs (that's "Three-Letter Acronyms," as you learned in Chapter 10)? You'll definitely want to refer back to the "DAQ and Other Data Acquisition Acronyms" section of Chapter 10, if you want to understand these next two paragraphs!

The Continuous Acquisition VI you just made takes advantage of NI-DAQmx's hardware timing

and memory handling capabilities. If, while your continuous acquisition task is running, an OS event ties up the CPU, NI-DAQmx will use the DAQ device's buffer and DMA capability (if any) to continue collecting data without any CPU involvement. DMA allows the DAQ hardware to write directly to the computer's memory, even if the processor is tied up. If the OS event interrupts the CPU longer than the on-board FIFO buffer and the DMA buffer can handle, only then will NI-DAQmx lose data samples.

This can best be understood by referring to the previous four snapshots of the buffer. First, assume the DAQ device has no on-board FIFO, but it has DMA capability. Ideally, the DAQ hardware will write data into the buffer continuously, and LabVIEW will be continuously reading a few samples behind the last sample written to the buffer. Suppose the CPU gets tied up after one second, and LabVIEW has read the first sample. When the CPU is tied up, the DAQ device can write to this buffer, but LabVIEW can't read from the buffer. If after twelve seconds, the CPU is still tied up, then LabVIEW has missed the data in the second slot from the left when it contained good data (light gray).

Streaming Data to a File

You have already written exercises that send data to a spreadsheet file. In those exercises, LabVIEW converted the data to a spreadsheet file format and stored it in an ASCII file after the acquisition completed. A different and sometimes more efficient approach is to write small pieces of the data to the hard disk while the acquisition is still in progress. This type of file I/O is called *streaming*. An advantage of streaming data to file is that it's fast, so you can execute continuous acquisition applications and yet have a stored copy of all the sampled data. Another reason to stream data to file is that if you are acquiring data continuously at a very fast rate for a long period of time, you will likely run out of memory.

Unless your data rate is very low (for example, less than 10 samples/sec), you normally will need to stream to binary, not ASCII, files. Binary files are much smaller in size and allow you to write data more compactly than is possible with text files.

With continuous applications, the speed at which LabVIEW can retrieve data from the acquisition buffer and then stream it to disk is crucial. You must be able to read and stream the data fast enough so that the DAQ device does not attempt to overwrite unread data in the circular buffer. To increase the efficiency of the data retrieval, you should avoid executing other functions, such as analysis functions, while the acquisition is in progress. Also, you can configure DAQmx Read to return raw binary data rather than voltage data (by selecting one of the raw data modes beneath More>>Raw>> from its Polymorphic VI Selector). This increases the efficiency of the retrieval as well as the streaming. When you configure DAQmx Read to produce only binary data, it can return the data faster to the buffer than if you used the analog waveform or voltage modes. One disadvantage to reading and streaming binary data is that users of other applications cannot easily read the file.

You can easily modify a continuous DAQ VI to incorporate the streaming feature. Although you may not have used binary files before, you can still build the VI in the following example. We will be discussing binary and other file types in Chapter 14.

Activity 11-8: Streaming Data to File

In this activity, you will modify Continuous Acquisition.vi, which you created in Activity 11-7, so

that it streams data to file.

1. Open Continuous Acquisition.vi, which you created in Activity 11-7, and save a copy as `Continuous Acquisition to File.vi`.

2. Modify the VI so that it looks like the front panel and block diagram in Figures 11.65 and 11.66.

Figure 11.65. Front panel of the VI you will create during this activity

[View full size image]

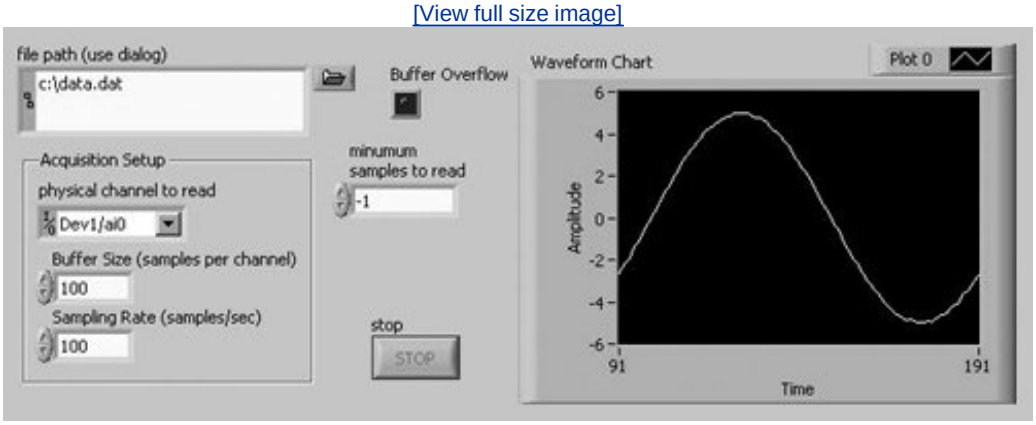
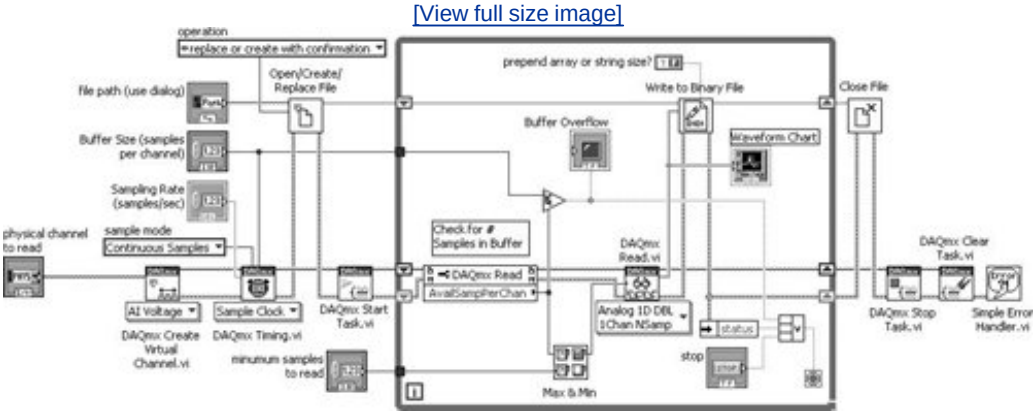


Figure 11.66. Block diagram of the VI you will create during this activity

[View full size image]



In this activity, you'll need to use the Write to Binary File function (from the Programming>>File I/O palette), which you learned about in Chapter 9, "Exploring Strings and File I/O." You'll learn more about reading and writing binary files in Chapter 14.

3. Use the Max & Min function to choose the max of points available or minimum points to read, as the number of data points to read by DAQmx Read. This will allow the software to run efficiently. (If the buffer already contains more samples than your threshold, it will be emptied in one large gulp. Otherwise, the program will wait until it can take the smallest nibble you specified. This results in the fewest cycles.)


Max & Min

4. Test for a buffer overflow by comparing whether the buffer size is less than or equal to the points available in the buffer. If a buffer overflow occurs, exit the While Loop.

5. Run this VI for a few seconds. You are now streaming data to a file!

To see the data written to disk, use the companion example VI Read Streamed Data File.vi, shown in Figures 11.67 and 11.68, found on the CD at `EVERYONE\CH11`. We've provided this VI for you because it involves a few concepts, such as type casting data, which you have not yet learned about. (You will learn about the Type Cast function in Chapter 14.)

Figure 11.67. Read Streamed Data File.vi front panel, used for reading the data files you create during this activity

[View full size image]

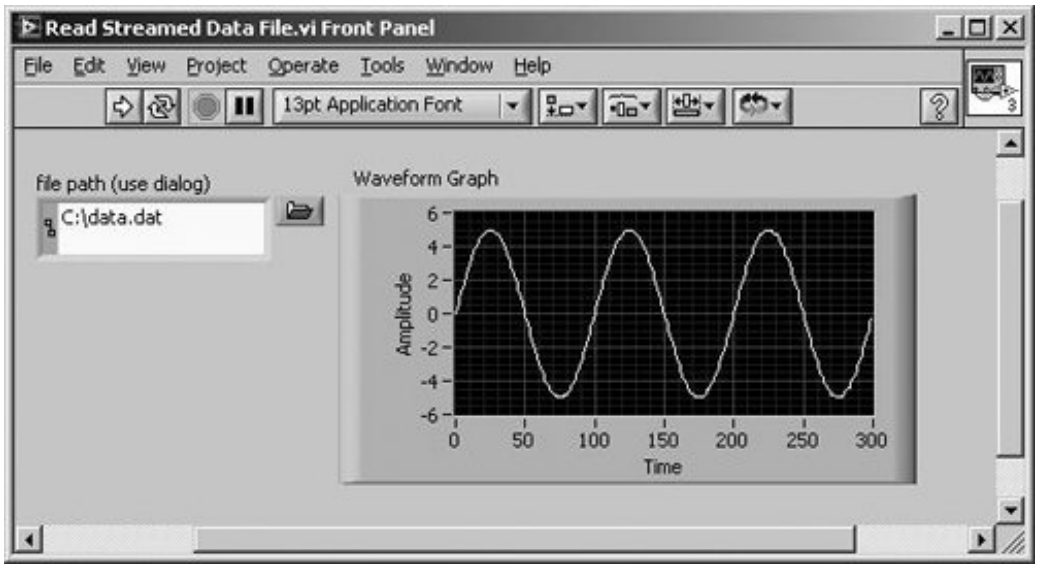
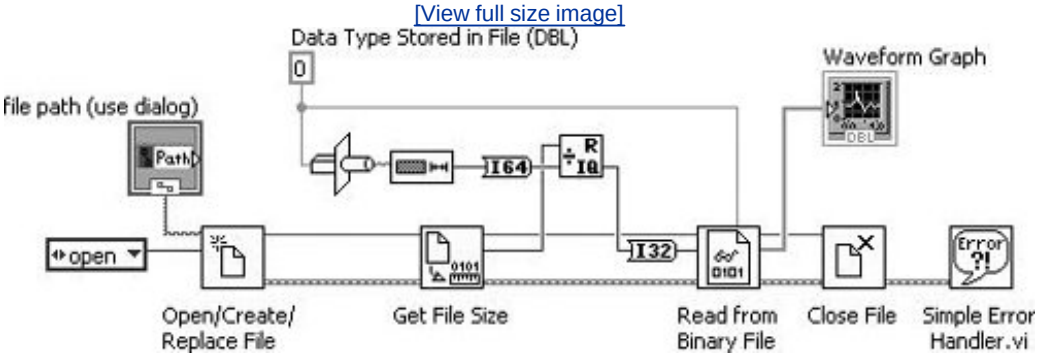


Figure 11.68. Read Streamed Data File.vi block diagram, used for reading the data files you create during this activity

[View full size image]



Run this VI, choosing the filename you used in the previous VI. Notice that the first VI, which streams data, doesn't attempt to graph the data. The point of streaming is to acquire data to the disk *fast*and look at it later.

Take a look at the VI's block diagram to see how it reads the binary data from the file (see Figure 11.68). Some of the stuff should make sense, but some of it might puzzle you. Don't waste time trying to figure out the puzzling stuff yet. We'll get to it later.

Counting Frequency and Events

As you know, digital signals can be either off or on, 0 or 1, true or false. Despite the fact that digital signals are bi-polar, they actually have a lot of interesting personality! Digital signals can have a wealth of *timing and event* characteristics. For example, how many times has a digital line changed

states, at what rate is it changing state, and for what amount of time has it been in the on state versus the off state? You can actually measure these digital signal characteristics and generate signals having the characteristics of your choosing!

However, trying to measure and generate digital signal timing and events using only software is very limited and complicatedit just doesn't work very well because software loops run slowly and unsteadily when compared to most digital signals. Fortunately, there is a special type of signal conditioning hardware for acquiring and generating digital timing and event signals: It is called the counter (or counter/timer).

A counter is an ASIC (Application Specific Integrated Circuit) computer chip that is designed for measuring and generating digital timing and events. It is likely that your multifunction DAQ device will have one or two counters on it. There also exist specific Counter/Timer Devices that have several counters (for example, 4, 8, or 16 counters) and several digital I/O lines but no analog I/O.

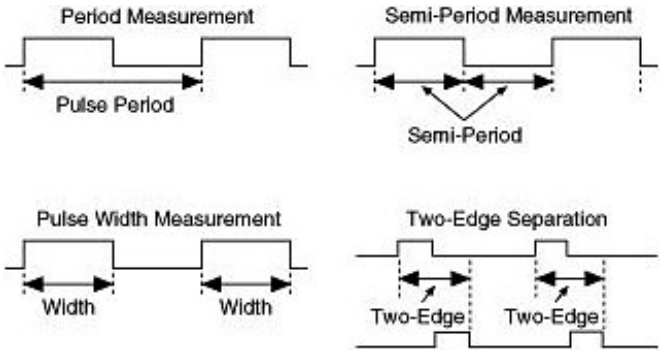
Measuring Digital Events and Timing with Counters

So, what kinds of counter measurements might you want to make? Table 11.3 lists common counter measurements, some of which are also illustrated in Figure 11.69.

Table 11.3. Counter Measurement Types

Measurement	***Description***
Edge Count	The number of pulses (rising or falling edges) that have occurred.
Period	The spacing between pulses.
Frequency	The rate at which pulses are occurring (1/period).
Pulse Width	The time that a pulse is active.
Semi-Period	The time that a pulse is either active or inactive (idle).
Two-Edge Separation	The time delay between the pulse edges of two signals.

Figure 11.69. Counter measurement types



Activity 11-9: Counting Digital Pulses

So, how do you count pulses in LabVIEW? In this exercise, you will see just how simple it is to count pulses with NI-DAQmx in LabVIEW.

For this activity, a simulated NI-DAQmx device will run, but it will not show pulse counts. You will need a physical NI-DAQmx device that has at least one counter channel, in order to count pulses.

1. Open a new VI.

2. Build the front panel and block diagram shown in Figures 11.70 and 11.71. All of the DAQmx VIs used in this example are found on the Measurement I/O>>DAQmx palette.

Figure 11.70. Front panel of the VI you will create during this activity

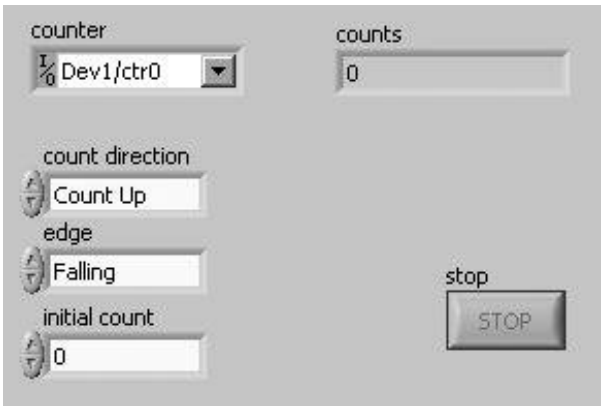
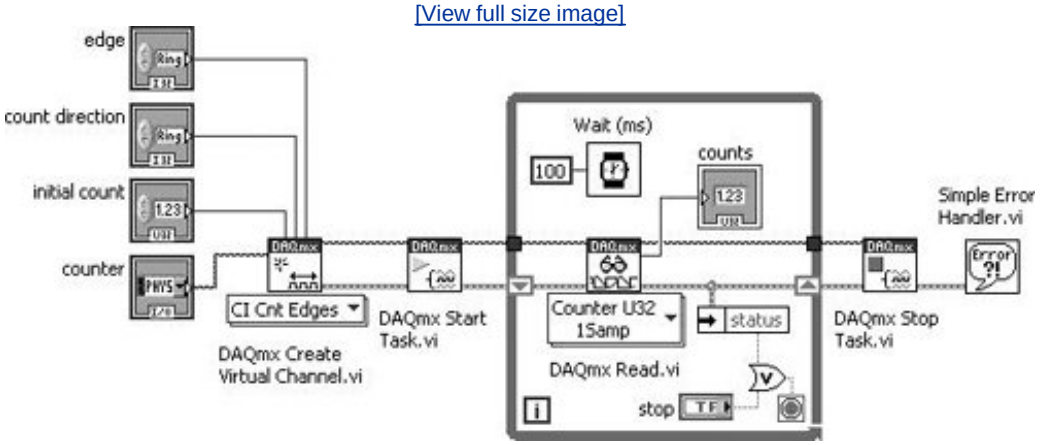


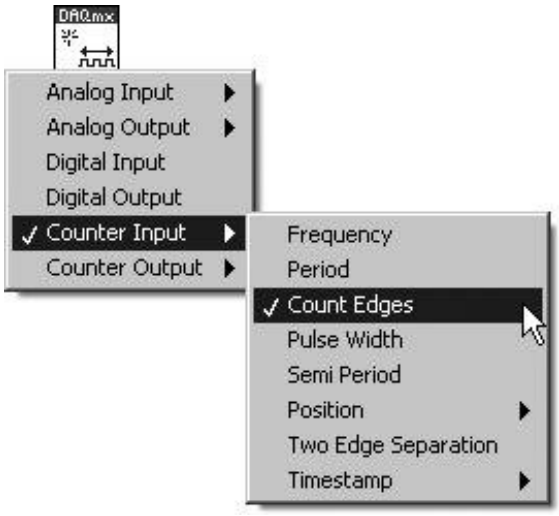
Figure 11.71. Block diagram of the VI you will create during this activity

[View full size image]



3. After placing DAQmx Create Virtual Channel onto the block diagram, select the Counter Input>>Count Edges instance from the Polymorphic VI Selector, as shown in Figure 11.72.

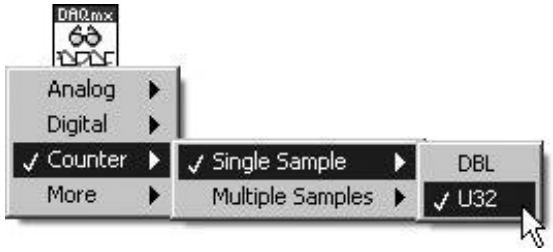
Figure 11.72. Selecting Counter Input>>Count Edges for DAQmx Create Virtual Channel



4. Create the counter, edge, count direction, and initial count front panel controls by right-clicking on the corresponding input terminals of DAQmx Create Virtual Channel and selecting Create>>Control from the pop-up menu.

5. After placing DAQmx Read onto the block diagram, select the Counter>> Single Sample>>U32 instance from the Polymorphic VI Selector, as shown in Figure 11.73.

Figure 11.73. Selecting Counter>>Single Sample>>U32 for DAQmx Read



6. Save your VI as `Count Input Pulses.vi`.

7. Now let's have some fun! Run your VI and start counting pulses. But, how do you generate pulses? Easysimply take a wire and connect it to the ground terminal (GND) of your DAQ device. Then, to generate a pulse, grab the other end of this wire and make it touch the PFI0 terminal (if you have selected CTR0otherwise, use PFI1 for CTR1, etc.). Grounding the counter input creates a pulse for as long as you keep it grounded.

Each time you touch the jumper wire to connect the terminals, you may see hundreds of counts. This is because the connection of the wire touching the terminal is not very good.

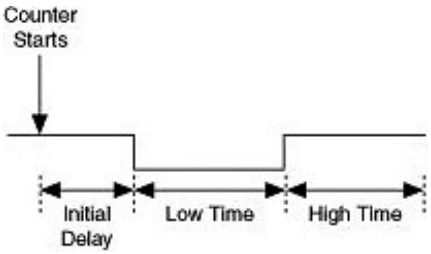
Digital Pulse Generation with Counters

Counters also allow you to generate *output* signals with the timing characteristics that you definethe same characteristics that a counter can measure, as just described. Table 11.4 contains the basic setup parameters for generating signals using your counter, some of which are illustrated in Figure 11.74.

Table 11.4. Pulse Setup Parameter Descriptions

Pulse Setup	***Description***
High Time	The time that the pulse is high, in seconds.
Low Time	The time that the pulse is low, in seconds.
Initial Delay	The time before pulse generation begins.
Idle State	The polarity of the generated signal: • HighThe terminal is at a high state at rest. Pulses move the terminal to a low state. • LowThe terminal is at a low state at rest. Pulses move the terminal to a high state.

Figure 11.74. Pulse setup parameter illustration



You can also specify how many pulses you wish to generate, or whether you want to generate pulses forever (or at least until you stop the NI-DAQmx generation task). This is called the Generation Mode, and the possible options are described in Table 11.5.

Table 11.5. Counter Signal Generation Modes

Generation Mode	*Description*
1 pulse	A single pulse.
N pulses	A train of N consecutive pulses. You specify the number of pulses to generate.
Continuous pulses	A train of continuous pulses.

Activity 11-10: Generating Digital Pulses

Experiment and try to create a VI that generates 1 pulse, another that generates N pulses, and another that generates continuous digital pulses. (Use the Digital Output>>Counter Output task type.)

Wire the counter output that you use for this activity into the counter input you used for the last activity, "Counting Digital Pulses," and count the pulses that you generate.

PREV NEXT

Wrap It Up!

This chapter has helped get you started with using LabVIEW to perform data acquisition.

We looked at the concepts of buffering and triggering. Buffering is a method for storing samples temporarily in a memory location (the buffer), and is used for fast and accurate sampling rates. Triggering is a method to initiate and/or end data acquisition dependent on a trigger signal. The trigger can come from the software or from an external signal. We also learned about digital I/O and its terminology, like "port," "line," "direction," and "state."

NI-DAQmx Tasks provide a solid model for signal measurement and generation. We discussed using MAX to create and edit tasks, referencing those tasks in LabVIEW, and generating LabVIEW configuration and examples code directly from your tasks!

Using NI-DAQmx tasks in LabVIEW is easy with the DAQmx VIsnearly all DAQ applications will only need to use the same five or six VIs. This ease of use comes from the polymorphism of the DAQmx VIs. These VIs let you acquire one point (nonbuffered AI) or a whole waveform (buffered AI) from one or multiple channels. They allow you to perform buffered I/O, hardware triggering, and disk streaming. They also allow you to read from and write to the digital lines of a DAQ device. You can read or write either a single line or a whole port (four or eight lines, depending on the DAQ device).

We also learned some advanced DAQ techniques such as timing and triggering, multi-channel acquisition, continuous acquisition, and streaming to file. We explored the interesting world of digital events and timing using counters to count digital pulses and generate pulses.

12. Instrument Control in LabVIEW

Overview

This chapter will give you a deeper look at instrument control. Sure, virtual instruments are the future of instrumentation, but there are a whole lot of non-virtual instruments sitting on benches, and you will definitely find yourself wanting to control them from LabVIEW. And, sometimes there's no question about using an external instrument. If you need to perform mass spectroscopy, for example, we don't know of any mass spectrometer available as a card that plugs into your PC! We'll take a look at the various communications options for instruments: serial communications, Ethernet, and GPIB controllers. You'll learn about VISA, a framework for communicating with all kinds of external instruments. We'll go through the basic steps necessary to get you started with instrument communication in LabVIEW, and point you in the right direction to do your own more advanced instrument control programs.

Goals

- Understand the different ways to connect to your instruments
- Discover the GPIB interface
- Investigate serial communications
- Discover SCPI, a language for instrument communication
- Learn about VISA, a communication framework for instrumentation
- Configure and construct VISA resources
- Learn how to use the Instrument I/O Assistant wizard
- Learn how to find, download, and install instrument drivers
- Learn about VISA VIs for instrument control
- Examine some samples of serial communication

Key Terms

- GPIB
- SCPI
- Serial
- Instrument driver
- VISA
- VISA resource
- Instrument I/O Assistant
- USB

Instrumentation Acronyms

In Chapter 10, "Signal Measurement and Generation: Data Acquisition," we learned many new acronyms related to data acquisition and DAQ hardware. Instrumentation also introduces some new acronyms, which are listed next. Some of them you probably already know!

GPIB: General Purpose Interface Bus. Less commonly known as HP-IB (Hewlett-Packard Interface Bus) and IEEE 488.2 bus (Institute of Electrical and Electronic Engineers standard 488.2), it has become a world standard for almost any instrument to communicate with a computer. Originally developed by Hewlett-Packard in the 1960s to allow their instruments to be programmed in BASIC with a PC, now IEEE has helped define this bus with strict hardware protocols that ensure uniformity across instruments.

IVI: Interchangeable Virtual Instruments. A standard for instrument drivers (software that you can use to control external instruments) that can work with a wide variety of different instruments.

LXI: LAN eXtensions for Instrumentation. A standard proposed by the LXI Consortium (http://www.lxistandard.org) for an instrumentation platform based on industry standard Ethernet technology designed to provide modularity, flexibility, and performance to small- and medium-sized systems.

RS-232: Recommended Standard #232. A standard proposed by the Instrument Society of America for serial communications. It's used interchangeably with the term "serial communication," although serial communication more generally refers to communicating one bit at a time. A few other standards you might see are RS-485, RS-422, and RS-423.

SCPI: Standard Commands for Programmable Instrumentation. A standard proposed by the SCPI Consortium (http://www.scpiconsortium.org) that specifies a structure and syntax for communicating with instruments, using simple, intuitive ASCII commands.

USB: Universal Serial Bus, a standard bus on most PCs for connecting external peripherals.

VISA: Virtual Instrument Standard Architecture, a driver software architecture developed by National Instruments. Its purpose is to try to unify instrumentation software standards, whether the instrument uses GPIB, PXI, VXI, or Serial (RS-232/422/485).

Connecting Your Computer to Instruments

You've got a good a PC and a lab full of instruments that you are excited to control using LabVIEW. What next? Well, the first step is to know how you are going to communicate with your instruments. Are they GPIB, Ethernet, or serial? Most instruments support one, if not several, of these communication methods.

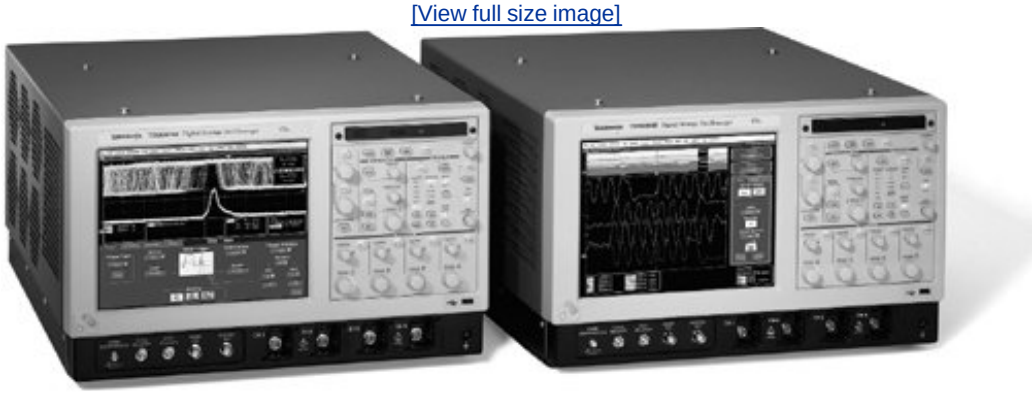
Using a GPIB Controller

A *GPIB controller* is a piece of hardware used to control and communicate with one or more external instruments that have the GPIB interface (see Figure 12.1). The General Purpose Interface Bus, invented by Hewlett-Packard (HP) in the l960s, became the most popular instrument communications standard for many decades. All HP (now Agilent) instruments support it, as well as thousands of others. GPIB was also updated and standardized by the IEEE, and was duly named IEEE 488.2. Some nice features about GPIB are the following:

- It transfers data in parallel, one byte (eight bits) at a time.
- The hardware takes care of handshaking, timing, and so on.
- Several instruments (up to 15) can be strung together on one bus.
- Data transfer is fast: 800 Kbytes/sec or more.

Figure 12.1. GPIB controllers are usually used to communicate with external bench-top instruments, such as oscilloscopes. (Image Copyright Tektronix, Inc. All Rights Reserved. Reprinted with permission.)

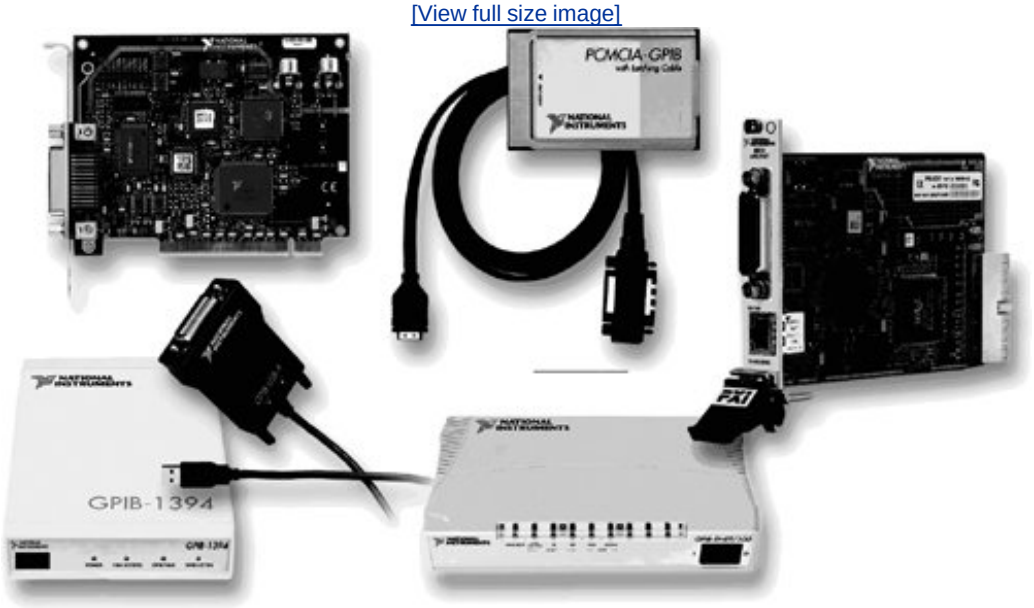
[View full size image]



You can obtain a GPIB controller for almost every platform and bus, as well as external interfaces that will convert your serial port, parallel port, Ethernet connection, or USB port to GPIB (see Figure 12.2).

Figure 12.2. GPIB controller devices come in many different interface flavors: PCI, PC Card, Ethernet, USB, PXI, and so on.

[View full size image]



Another feature that makes GPIB popular is that the computer and instrument talk to each other using simple, intuitive ASCII commands using the SCPI protocol, as described later in the section, "SCPI, the Language of Instruments."

A multitude of GPIB instrument drivers (LabVIEW VIs for controlling the instruments, which we will learn about later in the section, "Instrument Drivers") available free of charge, make communication with an external instrument as simple as using a few subVIs that do all the ASCII commands *for* you.

Installing a GPIB board is usually pretty straightforward. The GPIB boards from NI integrate with NI-MAX so that you can easily configure them under the Devices and Interfaces tree. The best advice is to just follow the installation procedures described in the manual that comes with the GPIB board.

Getting Ready for Serial Communications

Serial communication has the advantage that it's cheap and simple. Most industrial computers have at least one serial port available. However, for laptops and smaller desktops that do not have a built-in serial port, you can purchase an inexpensive USB-to-RS-232 adaptor (sometimes referred to as a "dongle") from companies such as Keyspan, IOGear, and Belkin. These generally provide from one to

four RS-232 ports, per adaptor. If you need many serial ports, consider a plug-in RS-232 board that can provide up to 16 RS-232 ports.

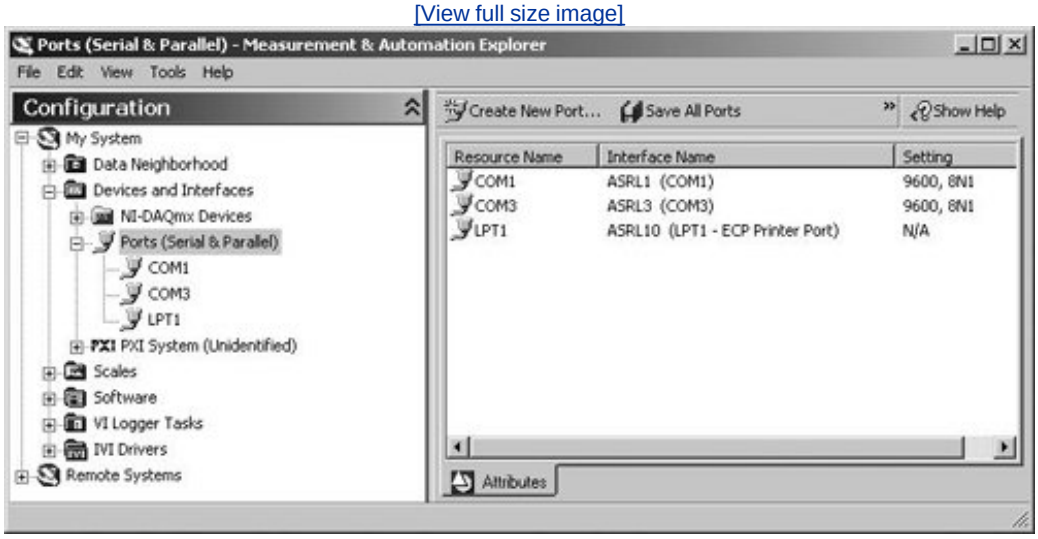
Even though it is easy to add serial ports to your computer and connect cables to your hardware, it is not always trivial to get the hardware hooked up and communicating correctly. Although popular standards such as RS-232 and RS-485 specify the cabling, connectors, timing, etc., manufacturers of serial devices more often than not abuse and ignore these standardsthe "RS" in RS-232 and RS-485 stands for "Recommended Standard," and the recommendations are not always followed. This means, unfortunately, that you'll need good documentation on the serial port settings from your instrument manufacturer, and more likely than not, will need to "tweak" the serial port settings on your PC.

Until all instrument vendors migrate to USB, you will find a lot of RS-232/485 instruments are still out there. If you need to use them, then it will be important to become familiar with your serial instrument, its pinouts, and the parameters it uses, such as baud rate, parity, stop bits, and so on.

Serial ports on a PC are referred to as COM1, COM2, and so on. You can see what serial ports are available on your computer, and how they are configured, by going to Measurement & Automation Explorer (MAX) and clicking on Devices and Interfaces>> Ports. Figure 12.3 shows a PC with two serial ports (COM1 and COM3), as well as a parallel port.

Figure 12.3. MAX showing serial ports configured on a PC

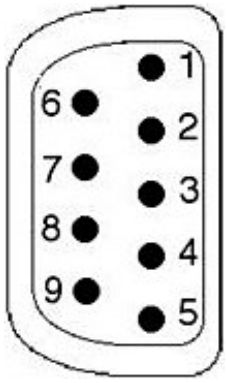
[View full size image]



In MAX, you can configure serial ports settings such as the baud rate, stop bits, and so on.

On the hardware side, it is very helpful to know what the different lines of a serial port are for. For example, it is not uncommon to build a custom serial cable. Also, some off-the-shelf cables do not have conductors for all the pins on the connectors. You will often need to "Ohm out" (perform a continuity test on) the cable to test the connections between pins on opposite ends of the cable. Figure 12.4 shows the serial pins on a DB-9 connector, the most common serial port configuration.

Figure 12.4. PC serial pinout (DB-9 connector) and pin signal assignments



Pin	*Name*	*Function*
1	DCD	Data Carrier Detect
2	RxD	Receive Data
3	TxD	Transmit Data
4	DTR	Data Terminal Ready
5	SG	Signal Ground
6	DSR	Data Set Ready
7	RTS	Ready to Send
8	CTS	Clear to Send
9	RI	Ring Indicator

In many cases, only a few of the lines are used. Chances are you'll end up using the following or fewer:

- Transmit (TxD): Sends data *from* the PC *to* the instrument.
- Receive (RxD): Sends data *to* the PC *from* the instrument.
- Signal Ground (SG): Ground reference. Never forget to hook this one up.
- Clear-to-Send (CTS): The PC uses this line to tell the instrument when it's ready to receive data.

- Ready-to-Send (RTS): PC uses this line to advise the instrument it's ready to send data.

If you experience problems getting your serial device to communicate, try a few of these things:

- Swap the transmit and receive lines. You can use a special cable, called a "null modem" cable, to do this or you can use a null modem adaptor, which is a small connector that is the same size and form-factor as a DB-9 or DB-25 gender-changer, but with identifying markings that state "null modem."
- Check the baud rate, parity, stop bits, handshaking (if any), and any other serial parameters on your PC. Then check what these parameters are on the instrument. If any of these differ, the two parties won't talk to each other.
- The serial instrument will always require a power supply. Make sure it's on.
- Make sure the CTS and RTS lines are connected properly. Some instruments require their use; others don't.
- Be sure the serial port is not being used by any other application.
- Check to see if you are using the serial port you think you're using (remember, there are usually at least two).
- Make sure you are sending the proper termination characters (EOL).

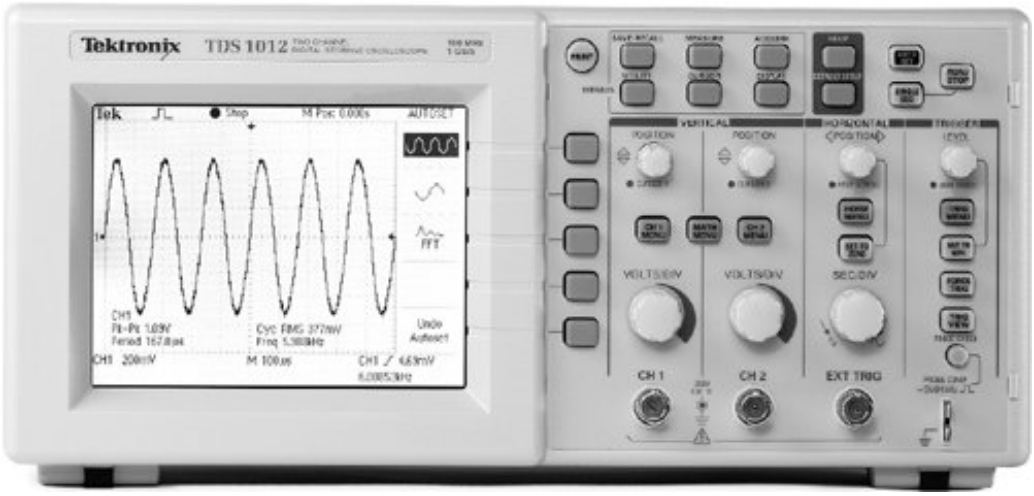
A handy way to know if your PC is set up correctly for serial communication is to connect two of its serial ports together using a null modem cable (or a regular cable plus a null modem adaptor). If you don't have two serial ports, you can add more ports using USB-to-RS232 dongles. Then, with a dumb terminal program (for example, HyperTerminal on Windows), you can verify that the data you type on one screen appears on the other and vice versa. Another cool trick is to use two *virtual* serial portsa pair of emulated serial ports that are connected (in software) by an imaginary null modem cable. There are several companies that sell virtual serial port software for a variety of operating systems.

Ethernet-Enabled Instruments

There are many instruments that have Ethernet ports and can be plugged into your network. They communicate using TCP/IP as the transport mechanism for the commands. Ethernet-enabled instruments are becoming popular and will continue to become more common, as time goes by (see Figure 12.5).

Figure 12.5. Ethernet-Enabled Tektronix TDS 1012 Digital Storage Oscilloscope (Image Copyright Tektronix, Inc. All Rights Reserved. Reprinted with permission.)

[View full size image]



Communicating with an Ethernet-enabled instrument is easy. As we will learn in the section, "VISA Resource Strings," you can address them in software using the following VISA resource string format:

```
TCPIP::host address[::LAN device name][::INSTR]
```

For example, if your device has an IP address of 10.0.1.124, and the instrument device is "SPX01," your VISA resource string can be either

```
TCPIP::10.0.1.124::INSTR
```

or

```
TCPIP::SPX01::INSTR
```

And that's all there is to it!

The LAN eXtensions for Instrumentation (LXI) standardan Ethernet-based instrumentation platform proposed by the LXI Consortium (http://www.lxistandard.org)is growing in popularity and industry adoption. It is designed to provide modularity, flexibility, and performance to small- and medium-sized systems in R&D and manufacturing engineering applications for a variety of industries. Due to the ubiquitousness of Ethernet and TCP/IP networks, we expect this standard, or other Ethernet-based standards, to continue to grow

SCPI, the Language of Instruments

SCPI, or the *Simple Command Protocol for Instruments*, is a set of rules for communicating with instruments using simple, intuitive ASCII commands. For example, a PC connected to an HP 34401A Digital Multimeter (DMM) might say something like this:

PC:	`IDN?` [Identity?meaning, who are you?]
DMM:	HEWLETT-PACKARD,34401A,0,54-12-60"
PC:	`MEAS:VOLT:DC?` [Measure DC Voltage]
DMM:	+4.23789

This simple dialog is an example of SCPI. You can find more information on the SCPI protocol online at http://www.scpiconsortium.org. But don't worry; you don't have to learn the SCPI protocol. Many instruments have LabVIEW instrument drivers that allow you to control the instrument using high-level VIs, without having to know anything about the device's command-set or the SCPI protocol. You will learn more about instrument drivers shortly.

Not all instruments communicate using the SCPI protocol. Many serial and Ethernet instruments use protocols that are not as human readable because they are designed for efficiency. If you are unsure, read your instrument's programmer's manualmost have one, and if not, there will usually be a small section in the user manual describing the communication protocol and command set.

VISA: Your Passport to Instrument Communication

You have learned about the different ways that instruments can be physically connected to your computer and a little bit about how to speak their language. And, you have even learned how some instruments support different types of physical connections (for example, GPIB *and* RS-232 Serial), while still speaking the same language, SCPI, in either case. So wouldn't it be nice if the software you wrote to communicate with such an instrument didn't care about the type of physical connection you were using? Then you wouldn't have to write different software for each type of physical connection you might use to connect to your instrument.

This is what VISA, or *Virtual Instrument Software Architecture*, provides for you. It gives you a set of functions in LabVIEW for sending commands to instruments and reading responses back from them. With VISA, it generally does not matter how the instrument is connected to your computer.

VISA Resource Strings

All VISA needs to know in order to communicate with your instrument is the following:

1. How your instrument is physically connected.

2. Where it is located on that physical connection.

You tell VISA this information via a VISA resource string. Table 12.1 shows some syntax for constructing VISA resource strings.

Table 12.1. Physical Connection VISA Resource String Syntax

Physical Connection	*VISA Resource String Syntax*
VXI	VXI[board]::VXI logical address[::INSTR]
GPIB	GPIB[board]::primary address[::GPIB secondary address][::INSTR]
PXI	PXI[bus]::device[::function][::INSTR]
Serial	ASRL[board][::INSTR]
Serial	COM[port number]
TCP-IP	TCPIP[board]::host address[::LAN device name][::INSTR]
TCP-IP (raw)	TCPIP[board]::host address::port::SOCKET

Physical Connection	*VISA Resource String Syntax*
USB	USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]

For example, here are some VISA resource names and their meanings.

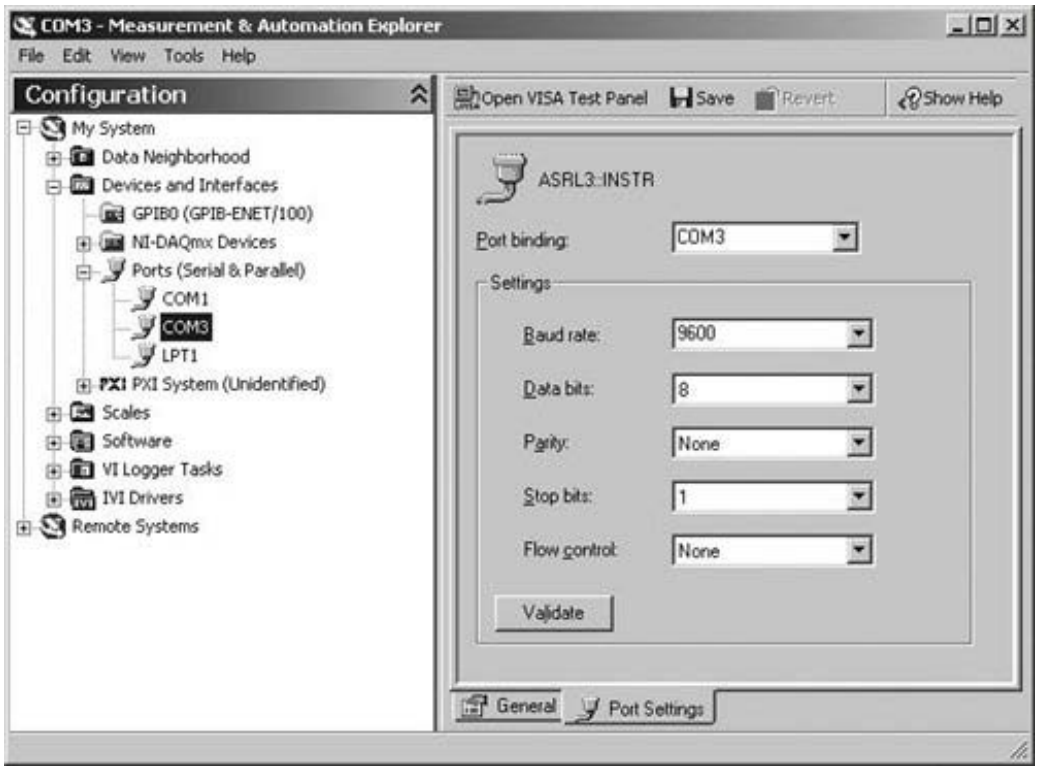
VISA Resource Name	*Description*
COM1	Serial port on COM1
ASRL3::INSTR	Serial port on COM3
GPIB0::12::INSTR	GPIB Board number 0, GPIB primary address 12

Configuring Your VISA Resources in MAX

You can configure your VISA resources in the MAX utility, beneath the Devices and Interfaces section, shown in Figure 12.6.

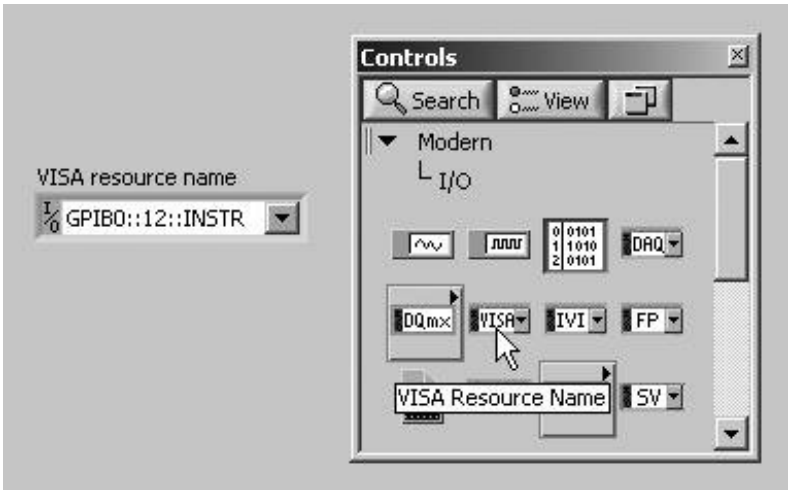
Figure 12.6. Configuring a serial port VISA resource's "Port Settings"

[View full size image]



Once your VISA resources are set up in MAX, you can communicate to your instrument in LabVIEW. In LabVIEW, you will use the VISA resource name control (found on the Modern>>I/O palette) to specify your VISA resource, as shown in Figure 12.7.

Figure 12.7. Placing a VISA Resource Name control from the palette onto a VI's front panel



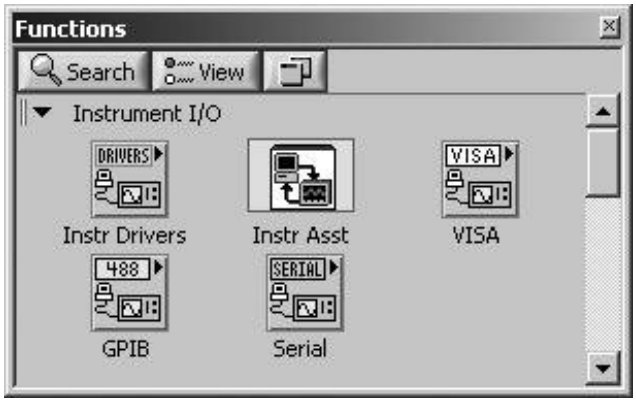
Now that you've seen how to set up your instrument with VISA and how LabVIEW can know about your instrument with a VISA resource name, you're ready for communicating with your instrument. You'll learn how to do this in the next section.

PREV NEXT

Instrument Control in LabVIEW

One of LabVIEW's great strengths is its ability to easily communicate with external instruments. The VIs on the Instrument I/O palette provide you with access to instrument drivers (libraries of VIs for communicating with specific instruments) that are installed, as well as the tools to create your own VIs that communicate with instruments directly (see Figure 12.8).

Figure 12.8. Instrument I/O palette



Before we get into the details of how to create VIs that communicate directly with instruments using command and response strings, let's first see how easy it is to communicate with instruments using the Instrument I/O Assistant.

Using the Instrument I/O Assistant

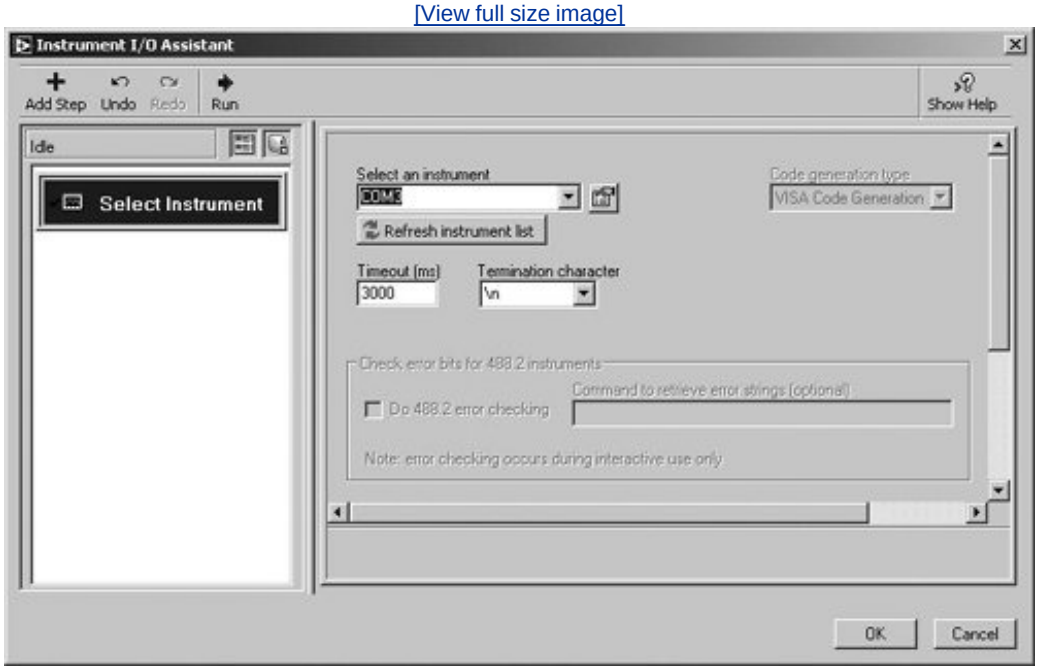
The Instrument I/O Assistant (shown in Figure 12.9) is an Express VI that allows you to very easily define a communication session with an instrument that is accessible via a VISA Resource. You can send commands, parse responses, and output the parsed data, all using the configuration dialog of the Instrument I/O Assistant Express VI.

Figure 12.9. Instrument I/O Assistant express VI

The Instrument I/O Assistant VI's configuration dialog (see Figure 12.10) allows you to select an instrument, by its VISA resource name, as well as communication timeout value and the termination character.

Figure 12.10. Configuration dialog of Instrument I/O Assistant

[View full size image]



From the configuration dialog, you can add steps that write commands, read and parse responses, and output the parsed data! When you are done configuring the Instrument I/O Assistant, you can use it in your VI, as is, or you can convert it into a regular subVI (for viewing and modifying its code) by selecting Open Front Panel from the Instrument I/O Assistant Express VI's pop-up menu.

If you remember the DAQ Assistant Express VI from Chapter 11, "Data Acquisition in LabVIEW," you'll see that this Instrument I/O Assistant is a similar high-level Express VI that will help you easily set up communication with your instrument.

Instrument Drivers

Your instrument is now connected to your computer and you are ready to start sending it commands and analyzing the data that it sends back to you. One of the easiest solutions to the problem of how to communicate with your instrument is to find an *instrument driver*. An instrument driver is a collection of LabVIEW VIs that hide the hard work of how the instrument communicates. LabVIEW instrument drivers simplify instrument programming to high-level commands, so you do not need to learn the mysterious, hard-to-remember, low-level, instrument-specific syntax needed to control your instruments. That way, instead of writing your own VIs that send commands like

```
:TRIG:SOUR IMM;:SAMP:COUN 10; READ? :SYST:ERR?
```

you can just use a "Read.vi" subVI, with the appropriate inputs, for your particular instrument (see Figure 12.11).

Figure 12.11. Sample Read VI from an instrument driver

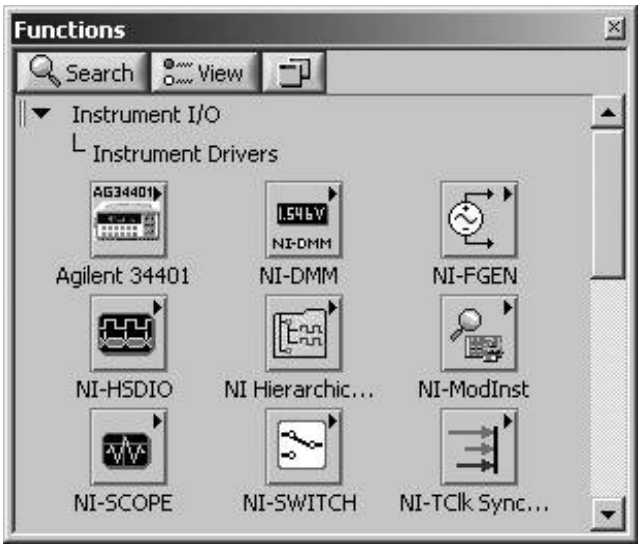
Instrument drivers are not necessary to use your instrument, of course (you can always use the low-level instrument commands if you wish), but they are merely time savers to help you develop your project so you do not need to study the instrument manual before writing a program.

Instrument drivers create the instrument commands and communicate with the instrument over Ethernet, serial, GPIB, or VXI. In addition, instrument drivers receive, parse, and scale the response strings from instruments into scaled data that can be used in your test programs. With all of this work already done for you in the driver, instrument drivers can significantly reduce development time.

With LabVIEW's ubiquitous presence in labs everywhere, most instrument manufacturers provide LabVIEW drivers that can either be found on their web sites or included on a software CD that comes with the instrument, if they aren't already on the NI web site or CD.

LabVIEW includes a couple of sample instrument driversfor example, the instrument driver for the Agilent 34401A Multimeter. If you look at the Instrument Driver palette (in the Instrument I/O palette), you can see what instrument driver functions look like (see Figure 12.12).

Figure 12.12. Instrument I/O>>Instrument Drivers palette



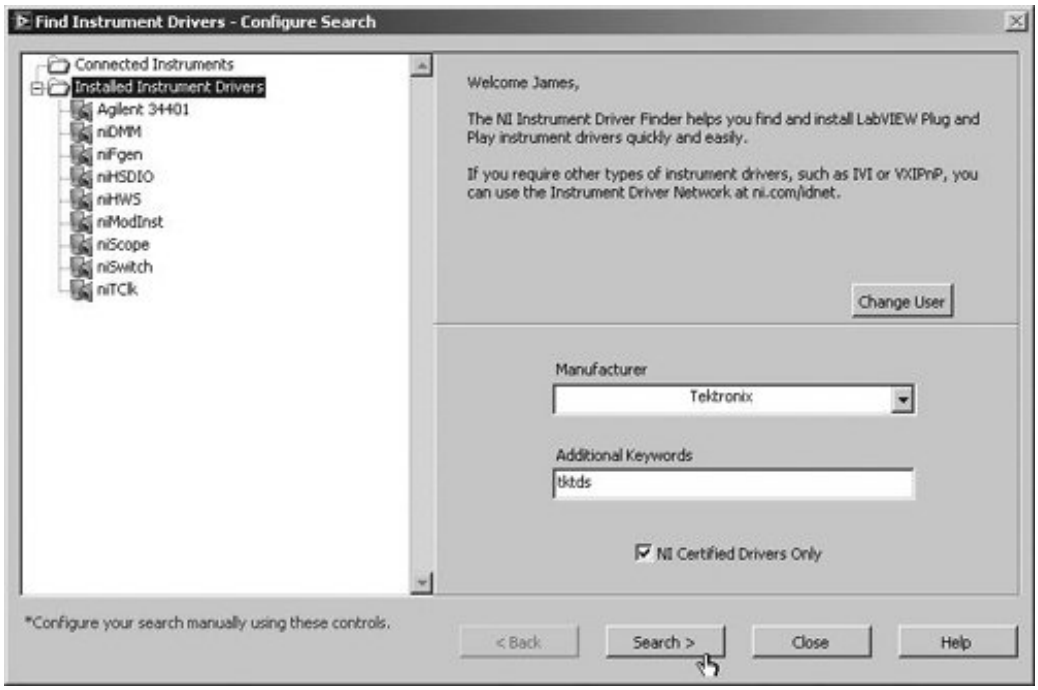
Although generally you'll need the instrument driver that is specifically for your instrument model, your odds are fairly good: LabVIEW provides more than 4000 LabVIEW instrument drivers from more than 200 vendors. A list is available on the *Instrument Driver Network* section of the National Instruments Developer Zone site: http://ni.com/devzone/idnet/. You can use these instrument drivers to build complete systems quickly. Instrument drivers will drastically reduce your software development time because you can practically just start using your instrument from LabVIEW. But, you don't have to go looking for instrument driversLabVIEW can find them for you, as you'll see in the next section!

Find Instrument Drivers from LabVIEW

Why spend your time searching the Web for instrument drivers, when you can have LabVIEW find and install them for you? The Find Instrument Drivers dialog (available from the Tools>>Instrumentation>>Find Instrument Drivers . . . menu) allows you to search for instrument drivers by manufacturer and keywords, as shown in Figure 12.13.

Figure 12.13. Find Instrument Drivers dialog showing installed instrument drivers

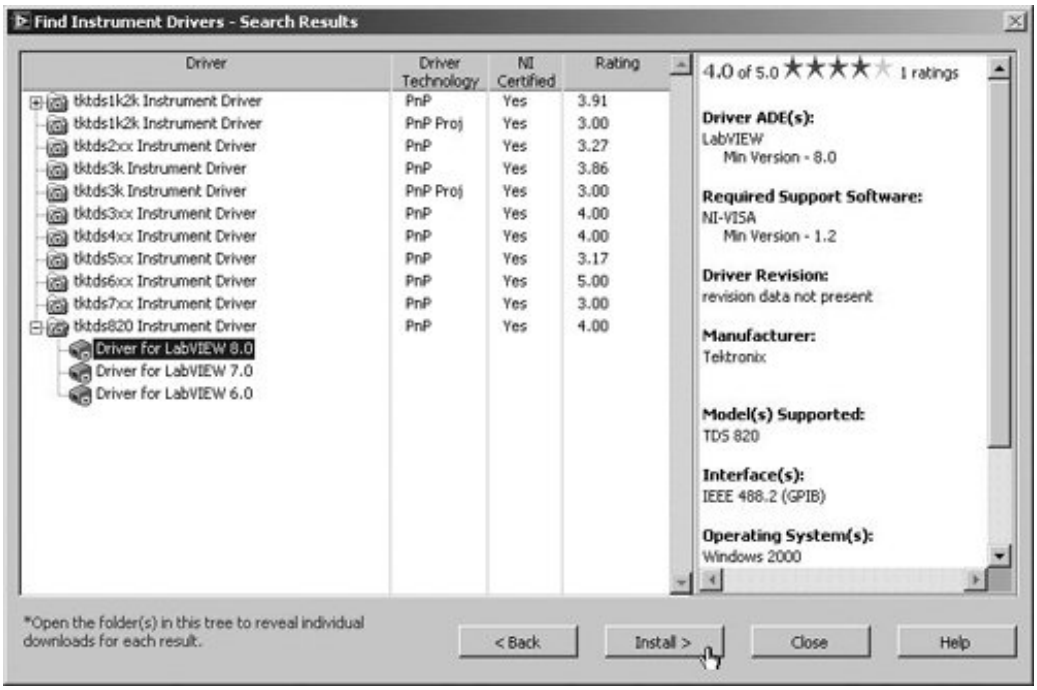
[View full size image]



The NI Instrument Driver Finder is the industry's largest source of instrument drivers. It provides access to instrument drivers for more than 4,000 instruments from over 200 different vendors. Just enter the instrument manufacturer name, model name, or other keywords, and press the Search button. Select the instrument model and LabVIEW version that you want to install and press the Install button, as shown in Figure 12.14.

Figure 12.14. Installing an instrument driver using the Find Instrument Drivers dialog

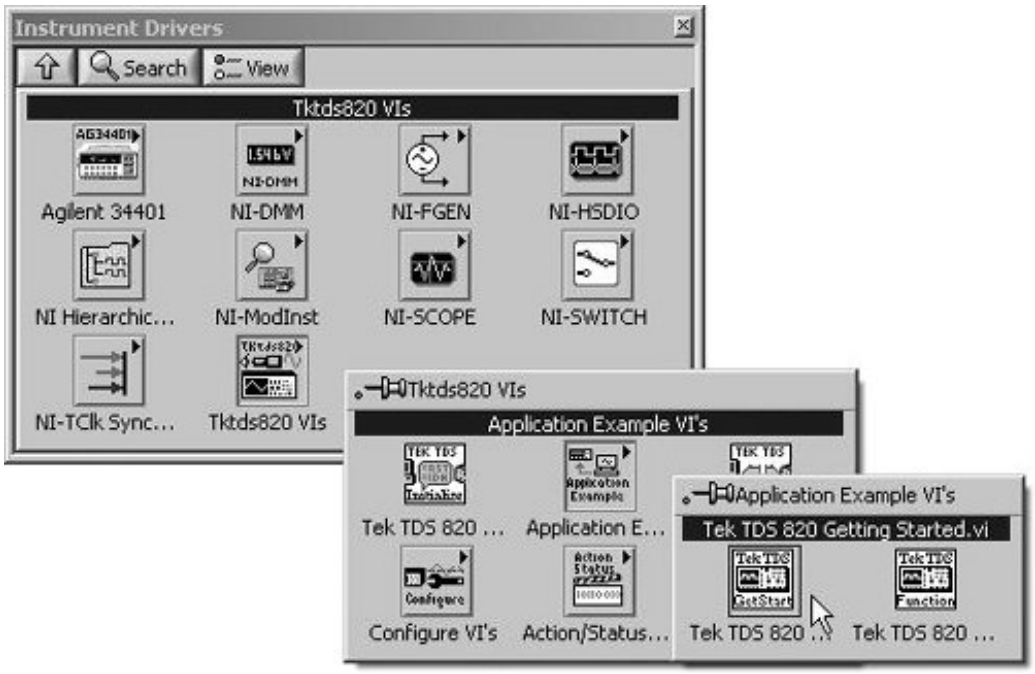
[View full size image]



LabVIEW will download the instrument driver directly into your Instrument Drivers palette (see Figure 12.15), and you will be controlling your instrument in minutesliterally!

Figure 12.15. A newly installed instrument driver, appearing in the palettes

[View full size image]

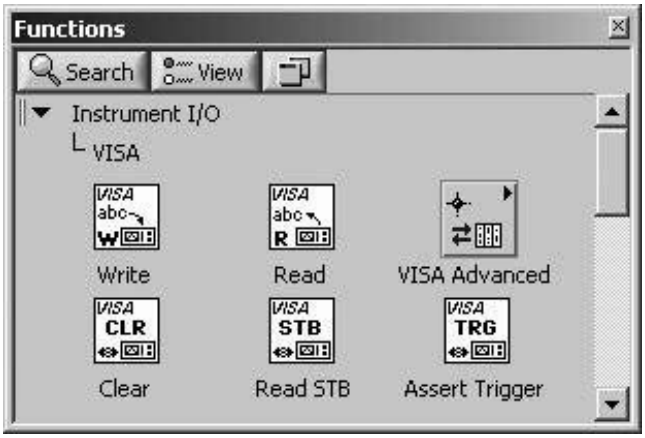


VISA Functions

As we mentioned earlier, VISA is a standard I/O Application Programming Interface (API) for instrumentation programming. Almost all instrument drivers for LabVIEW use VISA functions in their block diagrams. VISA can control VXI, GPIB, PXI, or serial instruments, making the appropriate driver calls depending on the type of instrument being used.

LabVIEW provides VISA functions in the Instrument Control>>VISA palette, shown in Figure 12.16.

Figure 12.16. Instrument I/O>>VISA palette



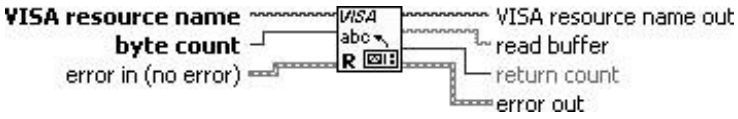
The VISA palette provides functions like VISA Write, VISA Read, CLR (clear), STB (status byte), and TRG (trigger). These last three are standard functions on most instruments.

VISA Write (Instrument I/O>>VISA palette) writes data to the device specified by the VISA Resource name (see Figure 12.17).

Figure 12.17. VISA Write

VISA Read (Instrument I/O>>VISA palette) reads data from the device (see Figure 12.18).

Figure 12.18. VISA Read



VISA Clear (Instrument I/O>>VISA palette) clears the device (see Figure 12.19).

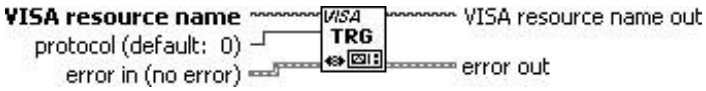
Figure 12.19. VISA Clear

VISA Read STB (Instrument I/O>>VISA palette) reads the status byte of the device (see Figure 12.20).

Figure 12.20. VISA Read STB

VISA Assert Trigger (Instrument I/O>>VISA palette) asserts software or hardware trigger on the device (see Figure 12.21).

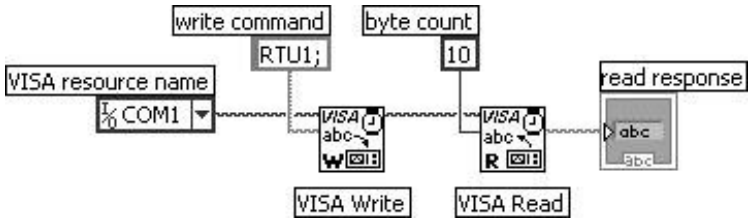
Figure 12.21. VISA Assert Trigger



The VISA resource name control is similar to a DAQmx name control; it tells the VISA functions what instrument (or "resource") you are communicating with. You can set up VISA resource names in your Devices and Interfaces in the MAX utility. Then you can use a VISA Resource Name constant from the Modern>>I/O palette to specify the instrument.

For example, if you had a serial device connected on port 1 (COM1), and you wanted to write a command specific to that device and read the 10 bytes that it returns, your block diagram might look like Figure 12.22.

Figure 12.22. Using a Visa Resource Name constant to address a device on COM1



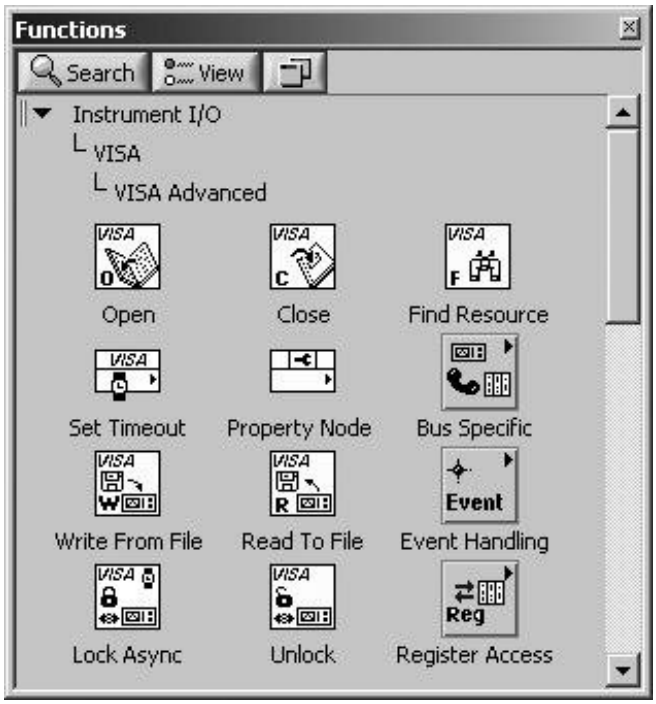
Of course, the command strings and lengths depend on the instrument's own particular protocol.

Occasionally, instrumentsdon't work very well with VISA communication, or you may have your own reason for not using VISAin any case, LabVIEW gives you the lower-level tools to communicate directly using serial or GPIB protocols.

Advanced VISA Functions

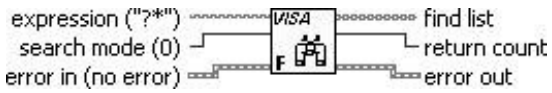
VISA has many advanced functions and settings, which are available from the Instrument I/O>>VISA>>VISA Advanced palette, shown in Figure 12.23.

Figure 12.23. VISA Advanced palette



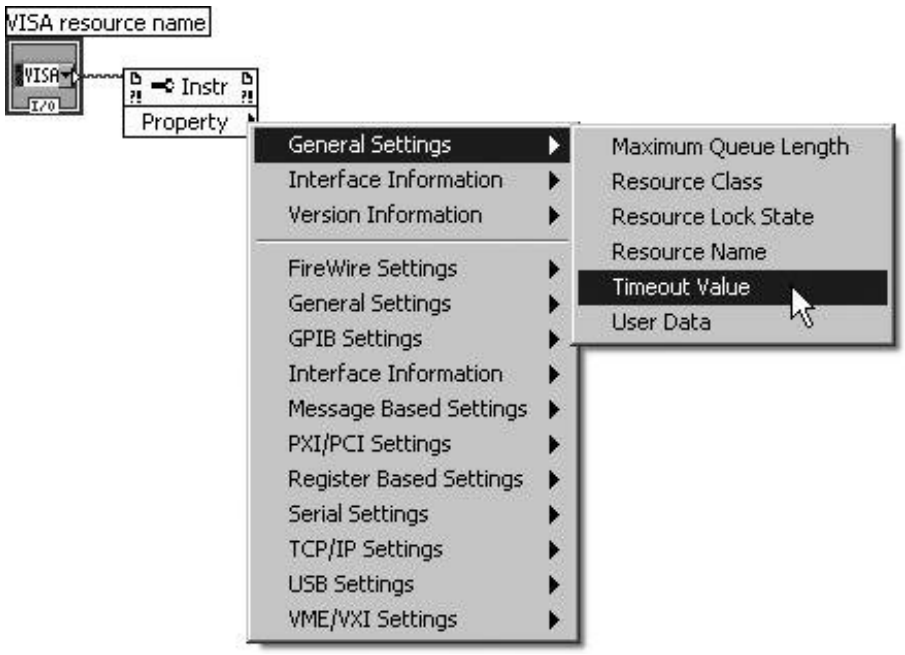
For example, there are functions for opening and closing VISA resources (those that are not persistently configured in MAX). There is a function called Find Resource (see Figure 12.24) that outputs a list of all VISA resources available on your system. This is useful for creating VIs that dynamically discover instruments that are connected to your computer!

Figure 12.24. Find Resource function



Another very useful feature of VISA is that you can use a Property Node to read and write properties of a VISA session. You can find the Property Node conveniently located in the Instrument I/O>>VISA>>VISA Advanced palette. Use the Operating tool to select the property you wish to read or write. Pop up on the node to select whether you wish to read or write the property. There are many properties to choose from. Many of the properties are bus (physical connection) specific, but the General Settings>> are applicable to all VISA sessions, such as the communication Timeout Value, shown in Figure 12.25.

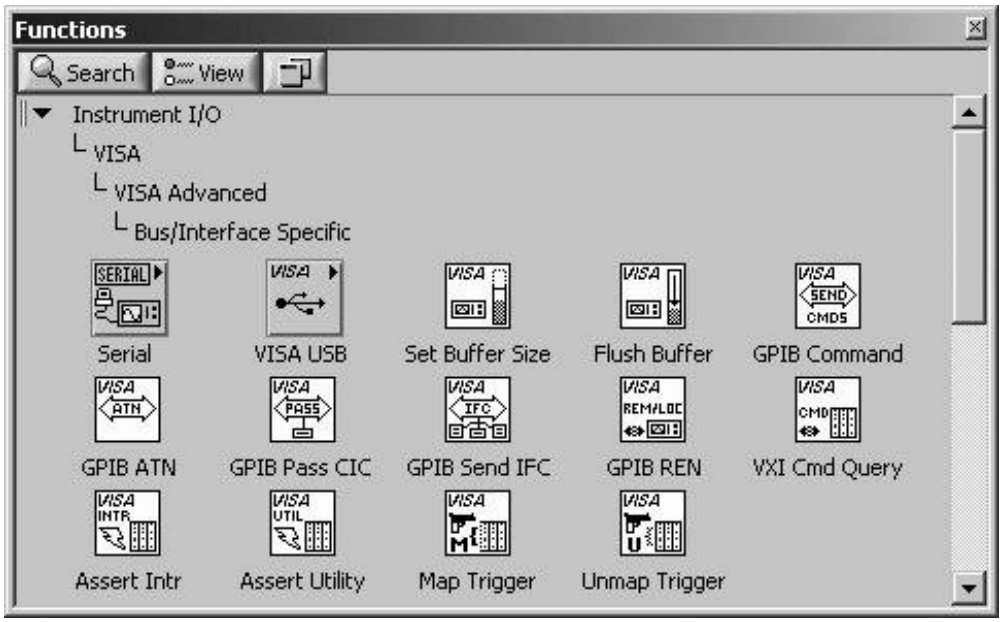
Figure 12.25. Configuring a property node to read a VISA session's Timeout Value



Bus/Interface Specific VISA Functions

VISA is a communication interface that supports instrument communication on a wide variety of physical connection typesfor a list, see Table 12.1 in the section, "VISA Resource Strings." The functions that you just learned about will work for all of them. However, there are some operations that are specific to the physical connection type, and these are located in the Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific palette, shown in Figure 12.26.

Figure 12.26. Bus/Interface Specific palette



Note that there are several GPIB functions, a Serial subpalette, and a VISA USB subpalette. We will see some of these discussed next.

VISA GPIB Versus Traditional GPIB Functions

For communicating with GPIB instruments, you have two choices: You can use VISA or you can use the *traditional* GPIB functions found beneath the Instrument I/O>>GPIB palette, which predate the VISA standard. *Unless you have a specific reason to use the traditional (non-VISA) functions for communicating with your GPIB-enabled instrument, use VISA!* It will make setup and communication with your device a lot simpler.

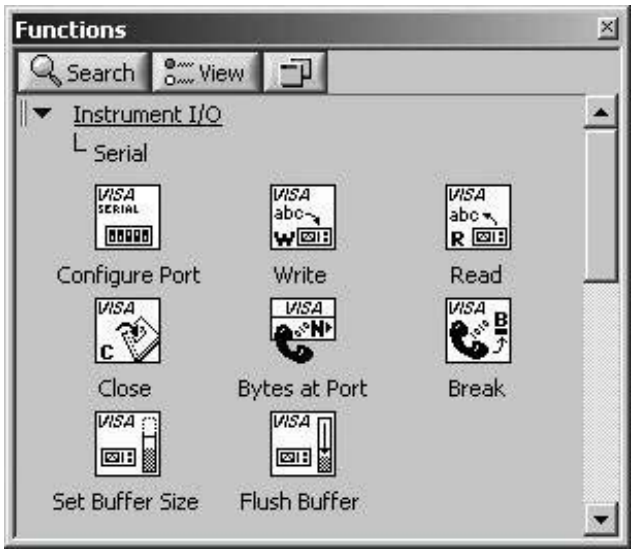
You should also note that there are some GPIB-specific VISA functions located in the Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific palette.

VISA Serial Functions

The Serial palette (see Figure 12.27) may be found in the Functions palette at the following two locations:

- Instrument I/O>>Serial
- Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific>>Serial

Figure 12.27. Serial palette



Certainly, the former is easier to find.

Notice that these functions are actually just specialized VISA functions (and VIs with VISA functions inside them) for serial ports. There are functions for reading and writing and, more specific to the serial port, you can do things like find out how many bytes are waiting to be read at the serial port, or configure the serial port (e.g., baud rate).

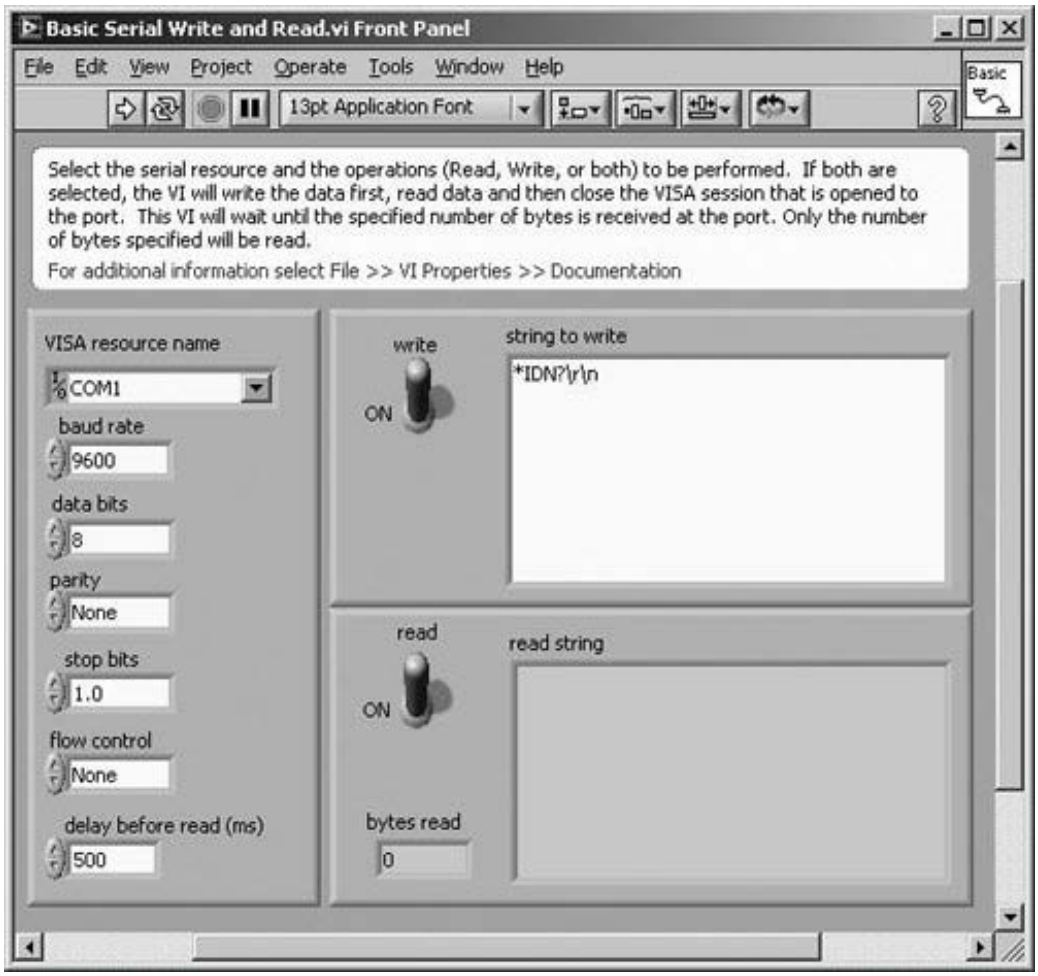
In one sense, serial communication is the simplest to program. On the other hand, serial communication suffers from abused and ignored hardware standards, obscure and complex programming protocols, and relatively slow data transfer speeds. The difficulties people encounter with writing an application for a serial instrument are rarely LabVIEW-related! We'll take a quick look at how to use the serial port VIs to communicate with a serial device.

You should become familiar with some basic concepts of how serial communication works if you've never used it before. If you have used a modem and know what things like baud rate, stop bits, and parity roughly mean, then you should know enough to get started. Otherwise, it might be a good idea to read some documentation about the serial port (any good book on RS-232).

A good place to get started, if you are talking to a serial instrument, is to use the Basic Serial Write and Read.vi example (see Figure 12.28) included with the LabVIEW examples. This VI, and other excellent serial port communication example VIs, may be found in the LabVIEW examples at `examples\instr\smplserl.llb`.

Figure 12.28. Basic Serial Write and Read.vi front panel

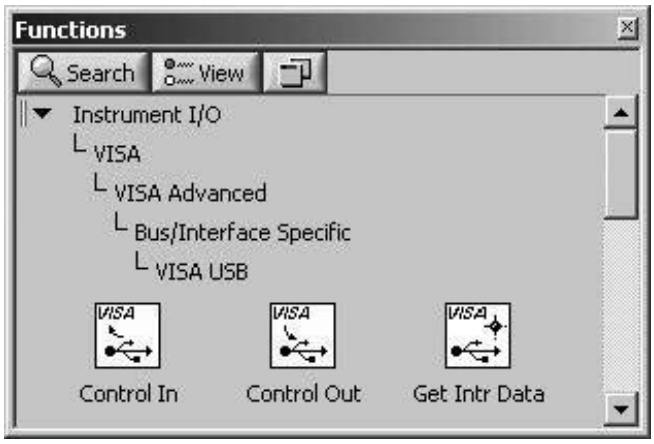
[View full size image]



VISA USB Functions

If you've haven't been hiding in a cave for the past few years, you already know that USB is quickly becoming the most popular communication bus for connecting devices to a PC. The VISA USB VIs and functions, found on the Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific>>VISA USB palette (see Figure 12.29), allow you to control USB devices.

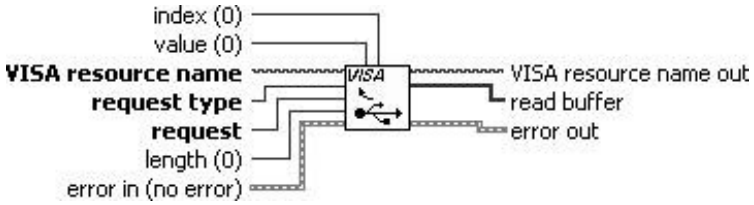
Figure 12.29. VISA USB palette



Here are descriptions of the VISA USB VI and functions.

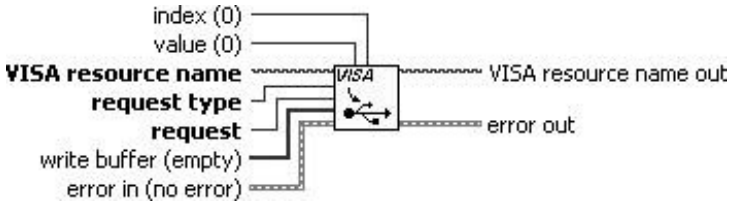
VISA USB Control In (Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific palette) performs a USB control pipe transfer from a USB device (see Figure 12.30). This function takes the values of the data payload in the setup stage of the control transfer as parameters. The function reads the optional data buffer read buffer if you require a data stage for this transfer.

Figure 12.30. VISA USB Control In



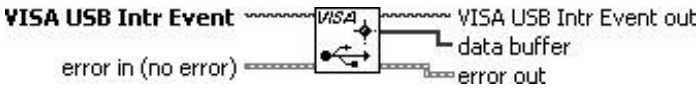
VISA USB Control Out (Instrument I/O>>VISA>>VISA Advanced>>Bus/Interface Specific palette) Performs a USB control pipe transfer to the device (see Figure 12.31). This function takes the values of the data payload in the setup stage of the control transfer as parameters. The function sends an optional data buffer write buffer if you require a data stage for this transfer.

Figure 12.31. VISA USB Control Out



VISA Get USB Interrupt Data (Instrument I/O>>VISA>> VISA Advanced>> Bus/Interface Specific palette) Retrieves the interrupt data that is stored in a VISA USB interrupt event (see Figure 12.32).

Figure 12.32. VISA Get USB Interrupt Data



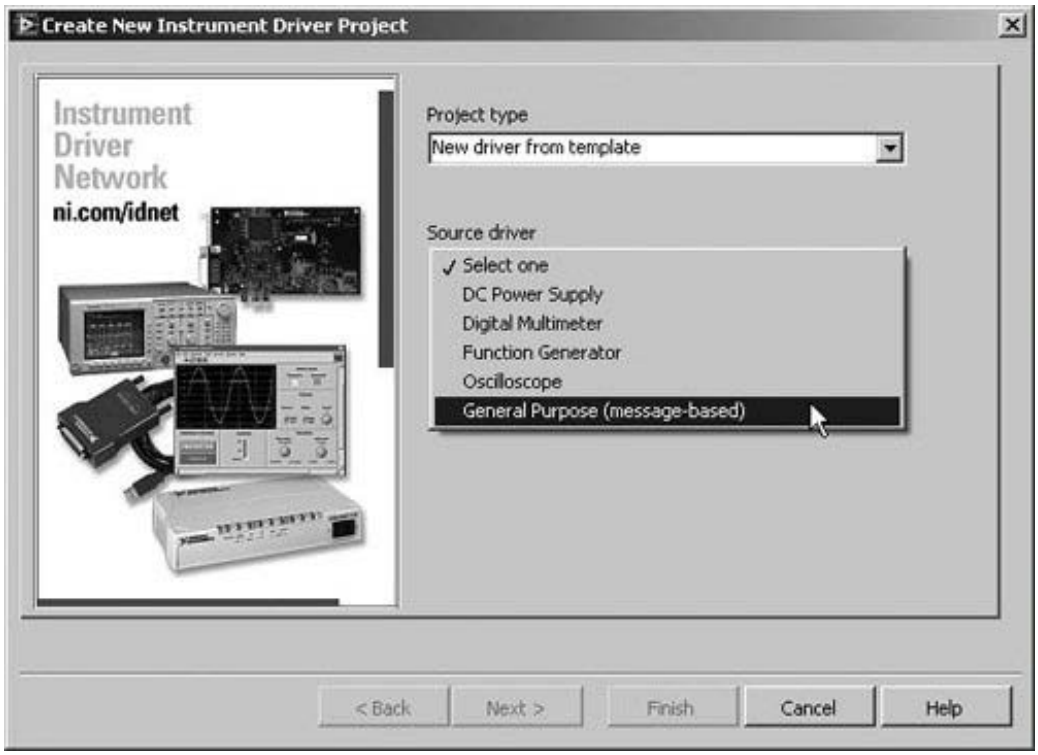
Create Your Own Instrument Driver: The Instrument Driver Wizard

If you would like to create your own instrument driver (because some instruments might not have an instrument driver already, or maybe you work for an instrument vendor), you can launch the Instrument Driver Wizard by selecting File>>New from the menu, selecting Project>>Project From Wizard>>Instrument Driver Project from the tree, and pressing the OK button.

You will be presented with a dialog (see Figure 12.33) that prompts you to answer several questions about the type of instrument driver you wish to create and allows you to specify several options.

Figure 12.33. Selecting an instrument driver type in the Instrument Driver Wizard

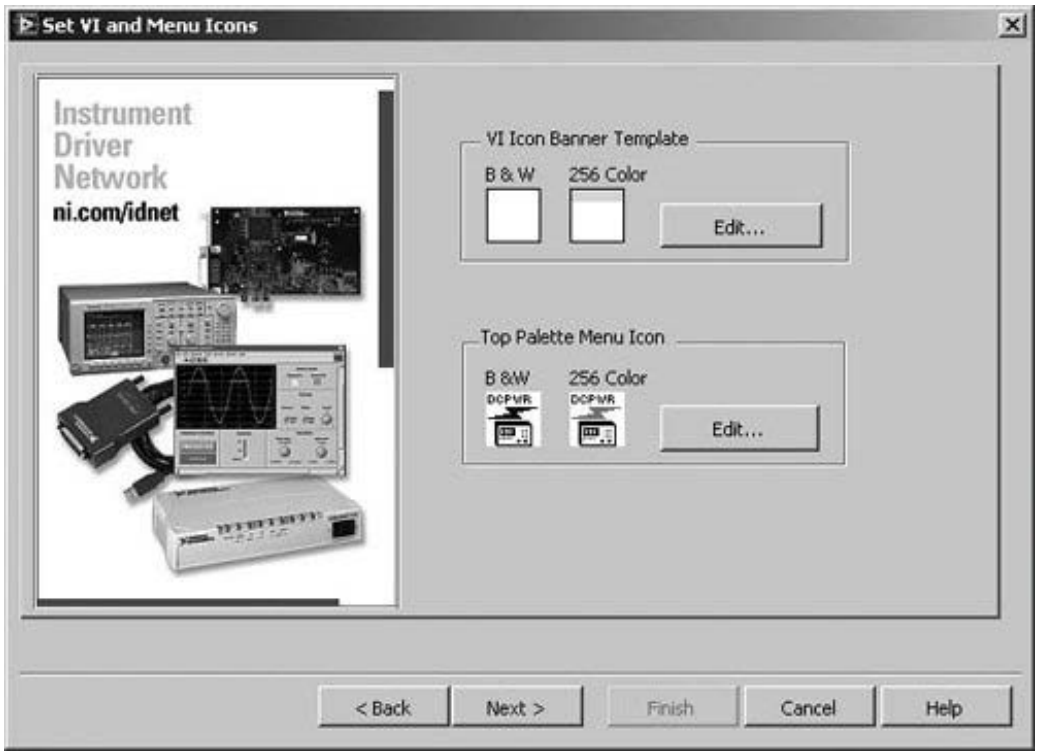
[View full size image]



You will be asked to give your instrument driver a name, and also to edit the VI Icon Banner Template and palette menu icon, as shown in Figure 12.34.

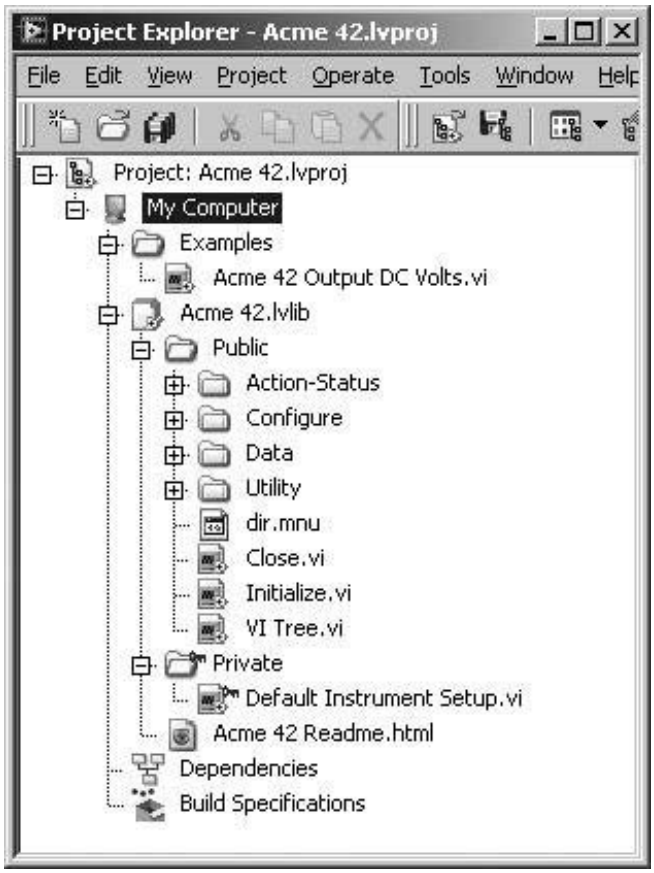
Figure 12.34. Editing the instrument driver icon in the Instrument Driver Wizard

[View full size image]



Once you have provided the Instrument Driver Wizard with all the necessary information, press the Finish button and a new LabVIEW Project will be created for your instrument driver, as shown in Figure 12.35.

Figure 12.35. Project Explorer window showing the newly created instrument driver



At the same time your new project is created, the LabVIEW Help documentation will be opened to the Controlling Instruments>>Using Instrument Drivers>>Instrument Driver Modification Instructions section, which provides instructions for modifying your instrument driver project. Make sure to read this documentation and follow the instructions provided.

PREV NEXT

Wrap It Up!

In this chapter, we learned the basics of instrument connectivity. We looked at some hardware aspects of communicating to external instruments through *GPIB*, *serial*, and *Ethernet* interfaces. And, we learned how to communicate with them quickly using the Instrument I/O Assistant Express VI, and how to use the VISA framework of instrument communication functions. These are all the tools you need to start communicating with your instruments from LabVIEW, today!

The GPIB interface is a widely accepted standard for many instruments, and you can often obtain an instrument driver in LabVIEW for your particular instrument. Serial communication is cheap and conceptually simple, but in practice requires much troubleshooting and tweaking. Ethernet is easyjust find a network cable, and you are ready to go!

SCPI, the language that many instruments speak, allows you to send simple ASCII commands to instruments, and the VISA instrument communication framework will get those commands to the instrument regardless of the hardware connection type. With VISA, you use LabVIEW to communicate with a plethora of instruments over a variety of connection types including GPIB, Ethernet/TCP/IP, Serial, USB, and many more.

13. Advanced LabVIEW Structures and Functions

Overview

This meaty chapter will show you how to use some of the more advanced and powerful functions and structures of LabVIEW. LabVIEW has the capability to manipulate and store local and global variablesmuch like conventional programming languages. You will also see how you can make controls and indicators more flexible and useful by using their property nodes, which determine the behavior and appearance of front panel objects. You'll learn to how to detect events in LabVIEW with the powerful Event Structure, for more efficient handling of your VI's GUI interaction. We'll take a look at type definitions and why they will save you time. Then you will study two bread-and-butter programming patterns in LabVIEW: state machines and queued message handlers. You will learn about the powerful messaging and synchronization functions, such as queues, notifiers, and more. Finally, we'll look at some miscellaneous advanced topics, such as how to "comment out" code, how to work with sound and graphics, and some cool GUI widgets.

Goals

- Understand local, global, and shared variables, and know how and when to use them (and, more importantly, when NOT to)
- Be able to customize the appearance and behavior of your front panel objects using property nodes
- Learn about invoke nodes
- Be able to use the Event Structure to detect GUI events and write event-driven code
- Know what a typedef is, and how to make one
- Become familiar with standard programming patterns like the state machine and the queued message handler
- See how messaging and synchronization functions, such as queues and notifiers, provide you with powerful and invaluable programming tools
- Incorporate two fundamental LabVIEW programming patterns into your coding: state machines and queued message handlers
- Learn how to disable, or comment out, code using the Diagram Disable Structure
- Discover the cool GUI controls and widgets, including graphics and sound

Key Terms

- Local variable
- Global variable
- Shared variable
- Race condition
- Read and write mode
- Property node
- Invoke node
- Event Structure
- Event Data Node
- Notify event
- Filter event
- Type definition (typedef)
- State machine
- Queued message handler
- Queue
- Notifier
- Semaphore
- Rendezvous
- Occurrence
- Diagram Disable structure
- Conditional Disable structure
- System controls
- Tab control
- Tree control

- Subpanels
- Splitter bars
- Scrollbars
- Drag and drop
- Sound
- Picture control

> *"Every program has at least one bug and can be shortened by at least one instructionfrom which, by induction, one can conclude that every program can be reduced to one instruction which doesn't work."*
>
> from an unnamed computer science professor

A word about this chapter: It's a whopper. It's the longest chapter in the book. But, there are many new, exciting, and important topics regarding LabVIEW covered here. This chapter is where you will learn the advanced topics that really start to uncover the power of LabVIEW. So dive in, but remember no one expects you to learn everything here in one sitting. Don't be intimidatedtake it one section at a time. Ready to learn the power tools in LabVIEW? Here we go.

PREV NEXT

Local, Global, and Shared Variables

Local and global variables are, technically speaking, LabVIEW structures. If you've done any programming in conventional languages like C or Pascal, then you're already familiar with the concept of a local or global variable. Up until now, we have read data from or written to a front panel object via its terminal on the block diagram. However, a front panel object has only one terminal on the block diagram, and you may need to update or read a front panel object from several locations on the diagram or other VIs.

Local variables (locals for short) provide a way to access front panel objects from several places in the block diagram of a VI in instances where you can't or don't want to connect a wire to the object's terminal.

Global variables allow you to access values of any data type (or several types at the same time if you wish) between several VIs in cases where you can't wire the subVI nodes or where several VIs are running simultaneously. In many ways, global variables are similar to local variables, but instead of being limited to use in a single VI, global variables can pass values among several VIs.

Shared variables are similar to global variables, but work across multiple local and networked applications. They also provide additional features that can buffer data and help avoid the synchronization problems that can be encountered with globals. VIs that use shared variables can run on all platforms, but can only be created on Windows.

This section will teach you some of the benefits of using locals, globals, and shared variables, as well as show you some common pitfalls to watch out for.

Local Variables

Local variables in LabVIEW are built-in objects that are accessible from the Programming>>Structures subpalette in the Functions palette (see Figure 13.1). When you select a local variable object, a node showing a "?" first appears to indicate the local is undefined. Clicking on this node with the Operating tool brings up a list of all current controls and indicators; selecting one of these will define the local. Or you can pop up on the local variable and choose Select Item to access this list (see Figure 13.2). You can also create a local variable by popping up on an object or its terminal and selecting Create>>Local Variable.

Figure 13.1. Local Variable structure, found on the

Programming>>Structures palette

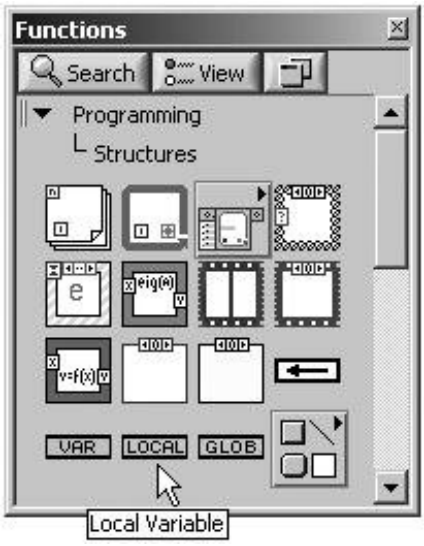
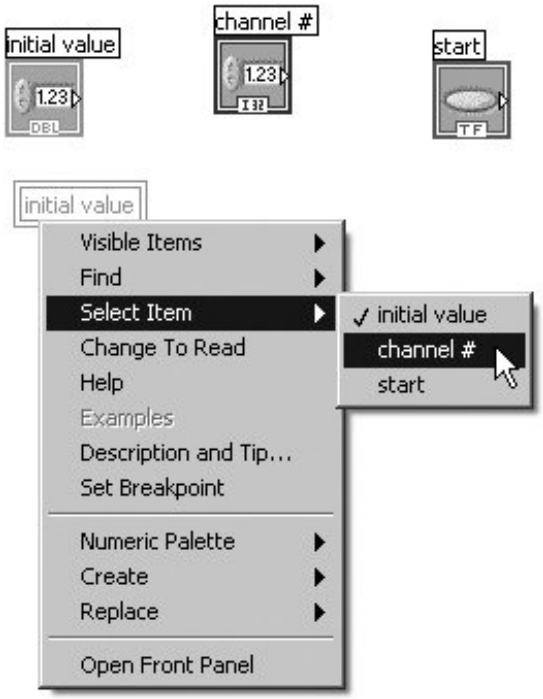


Figure 13.2. Associating a front panel control or indicator with the Local Variable structure on the block diagram



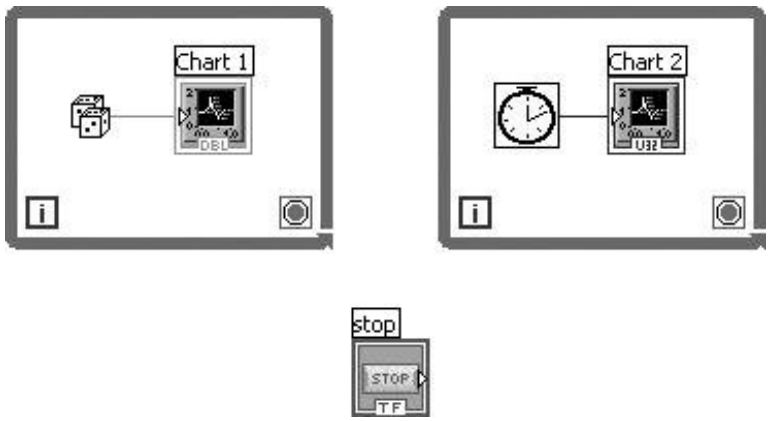
There are at least a couple of reasons why you might want to use locals in your VI:

- You can do things, such as control parallel loops with a single variable, that you otherwise couldn't do.
- Virtually any control can be used as an indicator, or any indicator as a control.

Controlling Parallel Loops

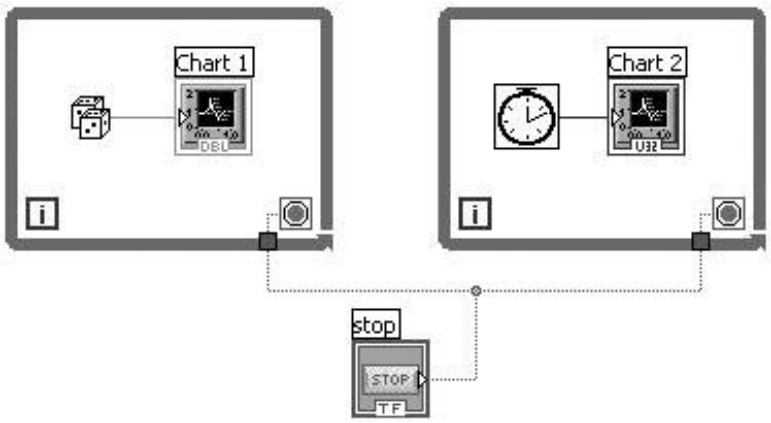
We've discussed previously how LabVIEW controls execution order through dataflow. The principle of dataflow is part of what makes LabVIEW so intuitive and easy to program. However, occasions may arise (and if you're going to develop any serious applications, the occasions *will* arise) when you will have to read from or write to front panel controls and indicators without wiring directly to their corresponding terminals. A classical problem is shown here: We want to end the execution of two independent While Loops with a single Boolean `stop` control (see Figure 13.3).

Figure 13.3. Two While Loops, but only one stop button (a dilemma)



How can we do this? Some might say we could simply wire the `stop` button to the loop terminals. However, think about how often the loops will check the value of the `stop` button if it is wired from outside the loops (see Figure 13.4).

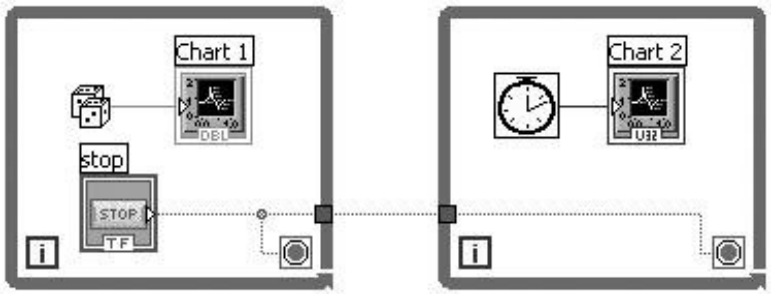
Figure 13.4. Wiring one stop button into two While Loops (this doesn't work)



Wiring the `stop` button from outside the loops to both conditional terminals will not work, because controls outside the loops are not read again after execution of the loop begins. The loops in this case would execute only once if the `stop` button is TRUE when the loop starts, or execute forever if `stop` is FALSE.

So why not put the `stop` button inside one loop, as shown in Figure 13.5? Will this scheme work?

Figure 13.5. A stop button inside one While Loop and wired into the next (this doesn't work, either)

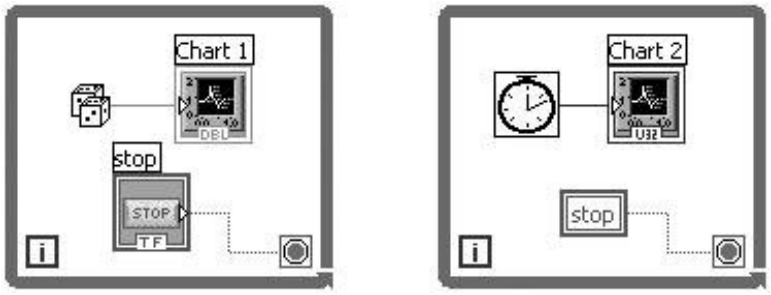


Putting the `stop` button inside one loop and stretching a wire to the other loop's conditional terminal won't do the trick either, for similar reasons. The second loop won't even begin until the first loop finishes executing (data dependency, remember?).

The solution to this dilemma isyou guessed ita local variable. Local variables create, in a sense, a "copy" of the data in another terminal on the diagram. The local variable always contains the up-to-date value of its associated front panel object. In this manner, you can access a control or indicator at more than one point in your diagram without having to connect its terminal with a wire.

Figure 13.6. A stop button inside one While Loop and a local variable

inside another (this works)



Referring to the previous example, we can now use one `stop` button for both loops by wiring the Boolean terminal to one conditional terminal, and its associated local variable to the other conditional terminal.

There's one condition on creating a Boolean local variable: The front panel object can't be set to Latch mode (from the Mechanical Action option). Although it isn't obvious at first, a Boolean in Latch mode along with a local variable in read mode produces an ambiguous situation. Therefore, LabVIEW will give you the "broken arrow" if you create a local variable of a Boolean control set to Latch mode. On the other hand, you can use local variables to mimic the Latch mode, by writing a value of TRUE to the button after it is read, as you'll learn next.

Blurring the Control/Indicator Distinction

One of the really nice features about local variables is that they allow you to *write to* a control or *read from* an indicator, which is something you can't normally do with the regular terminals of an object. Locals have two modes: *read* and *write*. A local variable terminal can only be in one mode at a time, but you can create a second local terminal for the same variable in the other mode. Understanding the mode is pretty straightforward: in read mode, you can read the value from the local's terminal, just as you would from a normal control; in write mode, you can write data to the local's terminal, just as you would update a normal indicator. Just remember this formula for wiring locals:

READ mode = CONTROL

WRITE mode = INDICATOR

Another way to look at it is to consider a local in read mode the data "source," while a local in write mode is a data "sink."

You can set a local variable to either read or write mode by popping up on the local's terminal and selecting the Change To. . . . . option. A local variable in read mode has a heavier border around it than one in write mode (just like a control has a heavier border than an indicator), as shown in Figures 13.7 and 13.8.

Figure 13.7. Local variable (write mode)

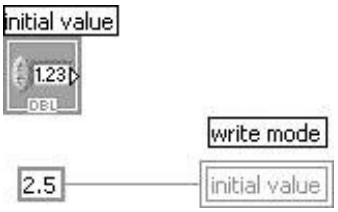
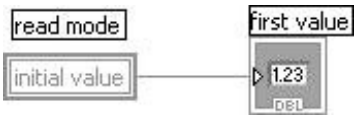


Figure 13.8. Local variable (read mode)



Pay close attention to these borders when you are wiring the locals, to make sure you are in the correct mode. If you attempt to write to a local variable in read mode, for example, you'll get a broken wireand it may drive you crazy trying to figure out why.

Last but not least, you must give a label to the control or indicator that the local refers to. That is, when creating a control or indicator, if you don't give it a name (which is possible in LabVIEW), you won't be able to create a local variable for it.

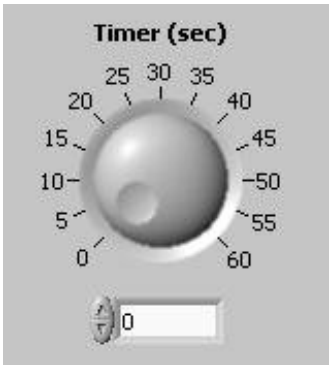
As a simple example, let's say you want to create a knob that represents a timer: The user can set the time, and watch the knob turn as it counts down the time, just like those old-fashioned timers used in the kitchen. Obviously, the front panel object is going to have to be a control because the user will set the time, but it must also be able to accept block diagram values and "turn" accordingly as time goes by. Try building the example shown in Activity 13-1.

Activity 13-1: Using Local Variables

In this activity you will create a simple "kitchen timer" VI that uses a local variable to update the timer's dial as it counts down to zero, behaving just like a real kitchen timer.

1. Build a front panel with a knob and select the Visible Items>>Digital Display option, as shown in Figure 13.9.

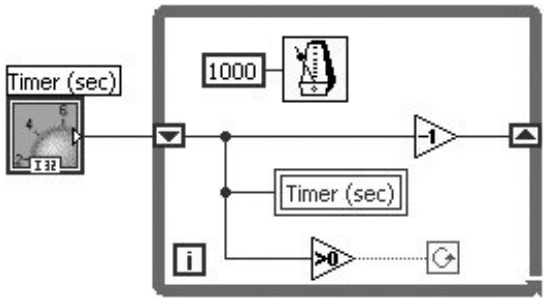
Figure 13.9. Front panel of the VI you will create during this activity



2. Create a local variable by selecting it from the Programming>>Structures palette. You will get a local variable icon. Click on this icon with the operating tool and select `Timer (seconds)`. The local should be in write mode by default.

3. Build the following simple block diagram (see Figure 13.10).

Figure 13.10. Block diagram of the VI you will create during this activity



4. Save your VI as `Kitchen Timer.vi`.

In this example, the shift register is initialized with the value of `Timer` that was set at the front panel. Once loop execution begins, the shift register's value is passed to the `Timer` local variable, which is in write mode, and then the shift register value is decremented. The Loop executes once per second, until the value counts down to zero. The knob on the front panel rotates to reflect the changed value.

A nice feature to add to this example would be a sound to alert you that the timer reached zero, just like the old-fashioned ones (Ding!). Later in this chapter, you will see how we can create simple

sounds with LabVIEW.

Locals sound like a great structure to use, and they are. But you should watch out for a common pitfall when using locals: race conditions. A race condition occurs if two or more copies of a local variable in write mode can be written to at the same time.

There is a hazard to using locals and globals: accidentally creating a *race condition*. To demonstrate, build this simple example shown in Figures 13.11 and 13.12.

Figure 13.11. Front panel of another VI you will create during this activity

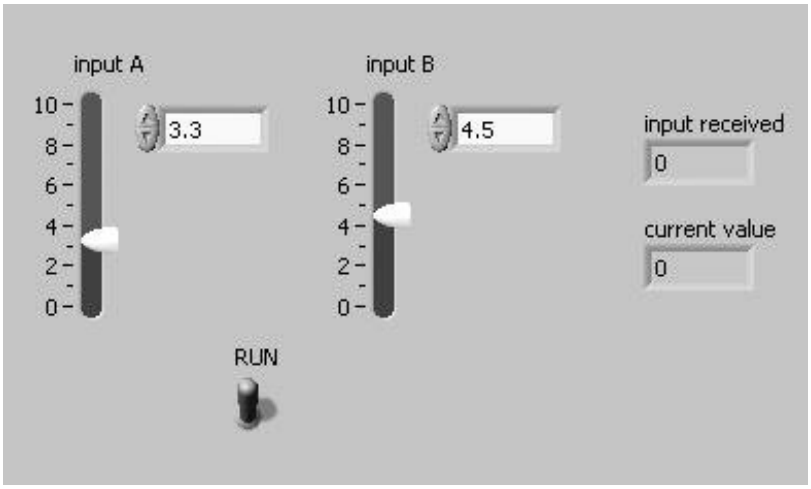
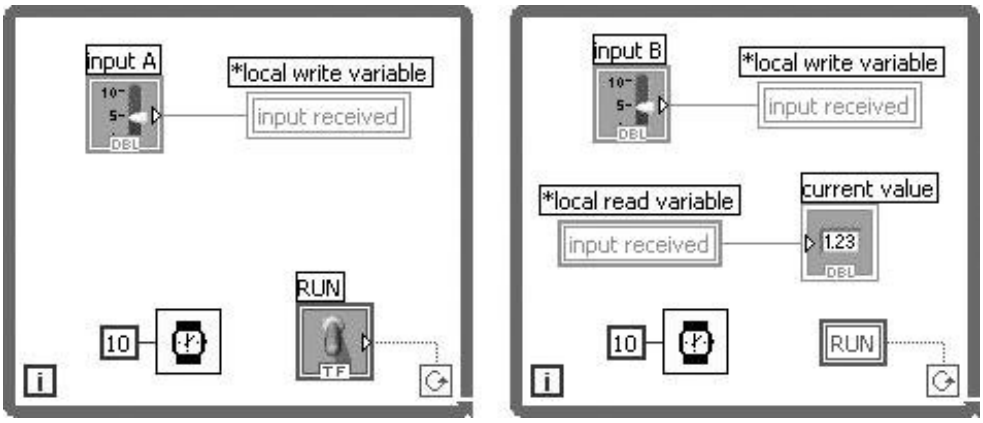


Figure 13.12. Block diagram of another VI you will create during this activity



Notice the two While Loops controlled by `RUN` and a local variable `RUN.` However, the local variable `input received` is being written in the left loop as well as in the right loop. Now, execute this VI with `RUN` set to FALSE and different values on the two sliders. The loops will execute just once. What value appears at `current value`? We can't tell you, because it could be the value from either `input A` or `input B`! There is nothing in LabVIEW that says execution order will be left-to-right, or top-to-bottom.

If you run the preceding VI with `RUN` turned on, you will likely see `current value` jump around between the two input values, as it should. To avoid race conditions such as this one, one must define the execution order by dataflow, sequence structures, or more elaborate schemes.

Another fact to keep in mind is that every read of a local creates a copy of the data in memory, a real problem for large arrays such as images. So when using locals, remember to examine your diagram and the expected execution order to avoid race conditions, and use locals sparingly if you're trying to reduce your memory requirement.

Activity 13-2: Fun with Locals

Another case where locals are very useful is in an application where you want a "Status" indicator to produce a more interactive VI. For example, you may want a string indicator that is updated with a comment or requests an input every time something happens in the application.

1. Build a simple data acquisition VI similar to the ones you wrote in Chapter 11, "Data Acquisition in LabVIEW," but modify it to have a string indicator that tells the user:

    - when the program is waiting for input.
    - when the program is acquiring data.
    - when the program has stopped.
    - if the program had an error, indicate it.

    To help you get started, Figures 13.13 and 13.14 show the front panel and block diagram.

Figure 13.13. Front panel of the VI you will create during this activity

[View full size image]

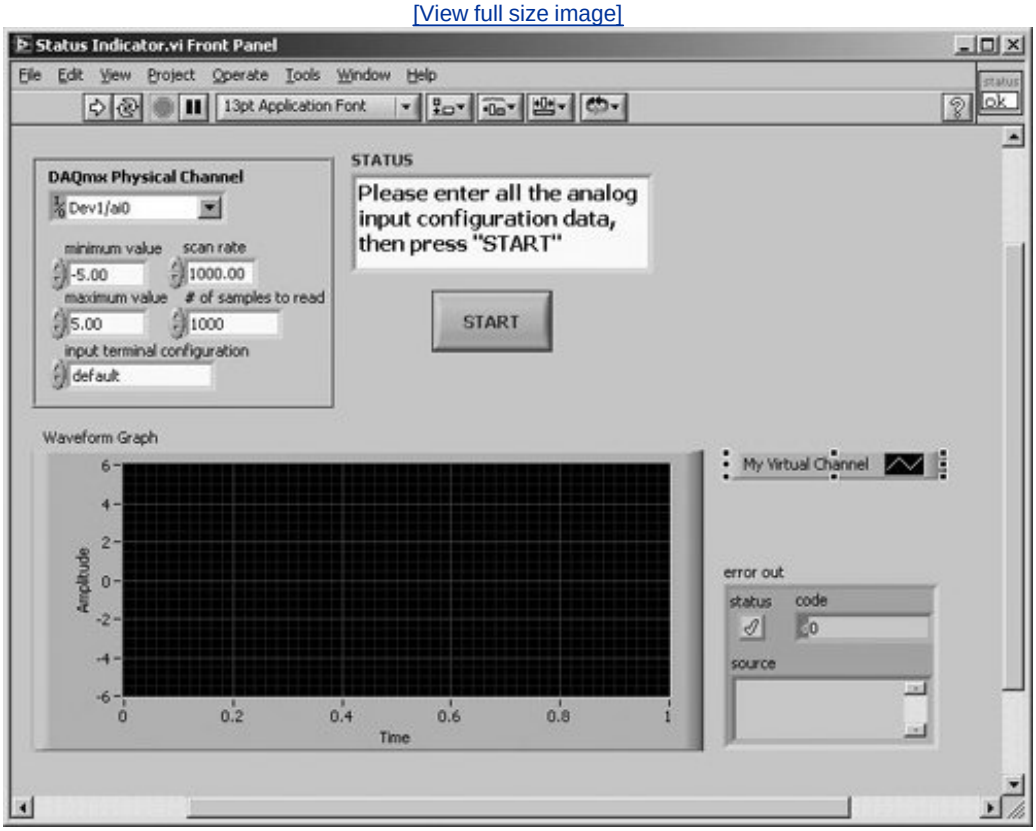
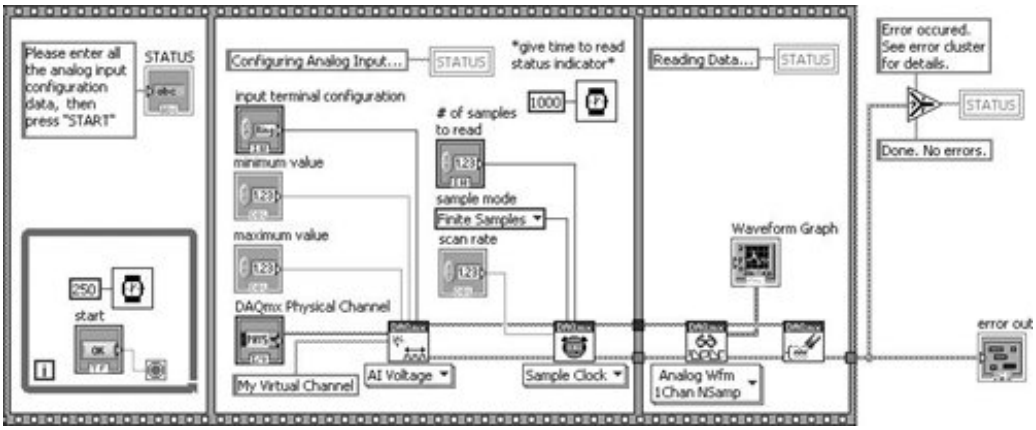


Figure 13.14. Block diagram of the VI you will create during this activity

[View full size image]



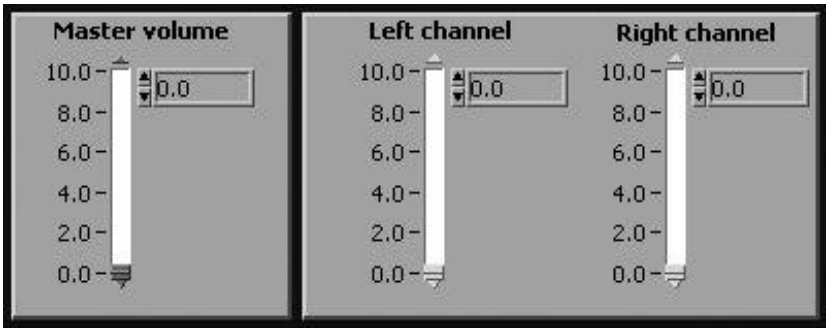
2. Save your VI as `Status Indicator.vi` in the `MYWORK` folder.

Activity 13-3: More Fun with Locals

In many applications, you may want some type of "master" control that modifies the values on other controls. Suppose you wanted a simple panel to control your home stereo volumes. The computer presumably is connected to the stereo volume control in some way. In the VI shown next, a simulated sound control panel has three slide controls: left channel, right channel, and master. The left and right channel can be set independently; moving the master slide needs to increment or decrement the left and right volumes proportionally.

Build the block diagram for the front panel shown in Figure 13.15. The fun part about this is that by moving the master slide, you should be able to watch the other two slides move in response.

Figure 13.15. Front panel of the VI you will create during this activity



Save your VI as `Master and Slave.vi`.

You will need to use shift registers for this exercise.

Global Variables

Global variables can be powerful and necessary, but before we even begin, beware. They are perhaps the most misused and abused structure in programming. We've seen many examples of bad programming because of an over-reliance on global variables. Globals are more often than not the cause of mysterious bugs, unexpected behavior, and awkward programming structures. Having said this, there are still occasions when you might want to and need to resort to globals. (Again, it's not that globals are badthey just need to be used with caution.)

Recall that you can use local variables to access front panel objects at various locations in your block diagram. Those local variables are accessible only in that single VI. Suppose you need to pass data between several VIs that run concurrently or whose subVI icons cannot be connected by wires in your diagram. You can use a global variable to accomplish this. In many ways, global variables are similar to local variables, but instead of being limited to use in a single VI, global variables can pass values among several VIs.

Consider the following example. Suppose you have two VIs running simultaneously. Each VI writes a data point from a signal to a waveform chart. The first VI also contains a Boolean `Power` button to terminate both VIs. Remember that when both loops were on a single diagram, we needed to use a local variable to terminate the loops. Now that each loop is in a separate VI, we must use a global variable to terminate the loops. Notice that the global terminal is similar to a local terminal except that a global terminal has a "world" icon inside it (see Figures 13.1613.19).

Figure 13.16. First VI front panel

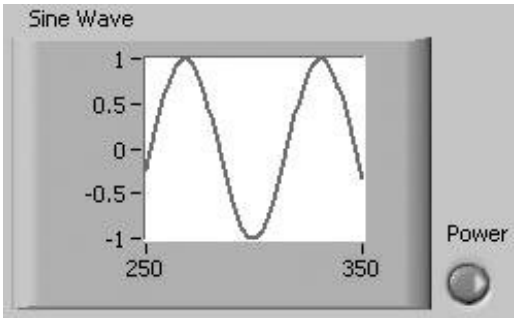


Figure 13.17. First VI block diagram

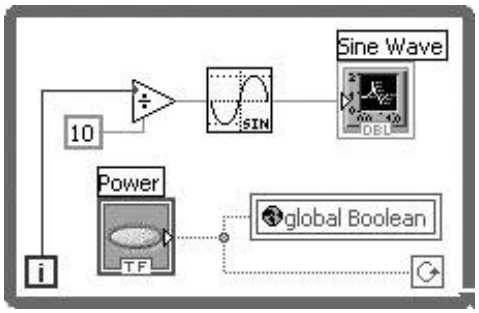


Figure 13.18. Second VI front panel

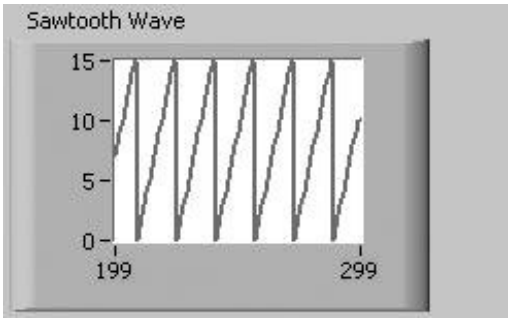
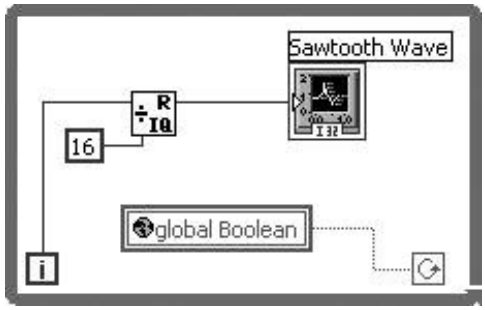


Figure 13.19. Second VI block diagram



Creating Globals

Like locals, globals are a LabVIEW structure accessible from the Programming>> Structures palette. And like locals, a single global terminal can be in write or read mode. *Unlike* locals, different VIs can independently call the same global variable. Globals are effectively a way to share data

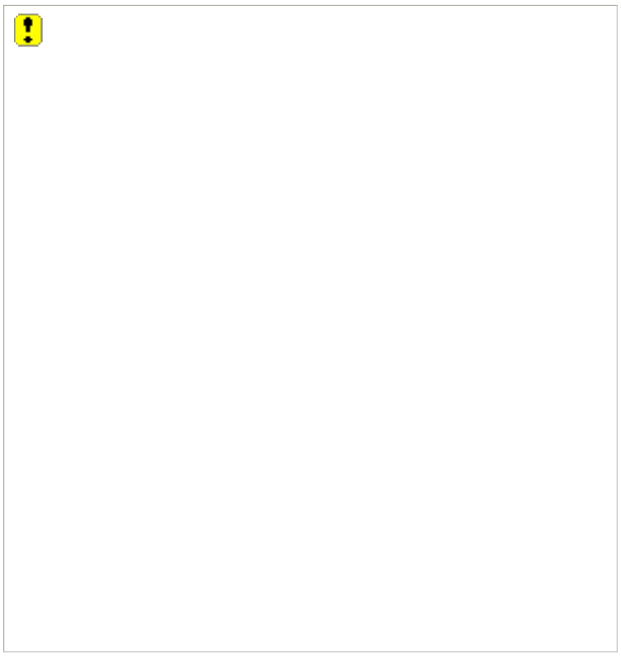
among several VIs without having to wire the data from one VI to the next; globals store their data independently of individual VIs. If one VI writes a value to a global, any VI or subVI that reads the global will contain the updated value.

Undefined Global Variable

Once the global structure is selected from the palette, an Undefined Global Variable appears on the diagram. The icon symbolizes a global that has not been defined yet. By double-clicking on this icon, you will see a screen pop-up that is virtually identical to a VI's front panel. You can think of globals as a special kind of VIthey can contain any type and any number of data structures on their front panels, but they have no corresponding diagram. Globals store variables without performing any kind of execution on these variables. Placing controls or indicators on a global's front panel is done in an identical fashion to a VI's front panel. An interesting tidbit about globals: It makes no difference whether you choose a control or an indicator for a given data type, because you can both read from and write to globals. Finally, be sure to give labels to each object in your global, or you won't be able to use them.

A global might contain, as in the following example, a numeric variable, a stop button, and a string control (see Figure 13.20).

Figure 13.20. A global VI front panel containing a few controls (which act as the global variables)



Save a global just like you save a VI (many people name globals with a "`_gbl`" suffix [for example, `myglobalvariable_gbl.vi`] just to help keep track of which files are globals and which are regular

VIs). To use a saved global in a diagram, you can choose Select a VI . . . in the Functions palette. A terminal showing one of the global's variables will appear on the diagram.

Operating Tool

To select the variable you want to use, pop up on the terminal and choose an item from the Select Item submenu, or simply click on the terminal using the Operating tool (see Figure 13.21). You can select only one variable at a time on a global's terminal. To use another variable, or to use another element in the global, create another terminal (cloning by <ctrl>-dragging or <option>-dragging is easiest).

Figure 13.21. Associating a global with a variable

Just like locals, globals can be in a read or a write mode. To choose the mode, pop up on the terminal and select the Change To . . . option. Read globals have heavier borders than write globals. As with locals, globals in read mode behave like controls, and globals in write mode act like indicators. Again, a global in read mode is a data "source," while a global in write mode is a data "sink."

READ mode = CONTROL

WRITE mode = INDICATOR

Some important tips on using global variables:

1. Only write to a global variable in one location, or carefully limit the number of locations, so that your global cannot be written to in more than one place at the same time. You may read from it in many locations. This convention avoids the problem of tracking down where a global variable's value is being changedyou only have one place to look. Also, it is best to limit the number of times that you write to a global variablewriting only once, during initialization of your application, is ideal.

2. Always, always initialize your globals in your diagram. The initial value of a global should always be cleared from your code by writing to them with initial values, before you read from them anywhere. Globals don't preserve any of their default values unless you quit and restart LabVIEW (or explicitly reset them to their default values).

3. Never read from and write to global variables in the same place; i.e., where either one could occur before the other (this is the famous "race condition").

4. Because globals can store several different data types, group global data together in one global instead of several global variables.

It's important that you pay attention to the names you give the variables in your globals. All the VIs that call the global will reference the variable by the same name; therefore, be especially careful to avoid giving identical names to controls or indicators.

An Example of Globals

Let's look at a problem similar to the two independent While Loops. Suppose that now, instead of having two independent While Loops on our diagram, we have two independent subVIs that need to run concurrently.

The subsequent figures show two subVIs and their respective front panels. These two VIs, Generate Time.vi and Plot.vi, are designed to be running at the same time. (Both of these VIs may be found on the CD-ROM, in the EVERYONE\CH13 folder.) Generate Time.vi just continuously obtains the tick count from the internal clock in milliseconds starting at the time the VI is run (see Figure 13.22). Plot.vi generates random numbers once a second until a stop button is pressed, after which it takes all the tick count values from Generate Time.vi and plots the random numbers versus the time at which the numbers were generated (see Figure 13.23).

Figure 13.22. Generate Time.vi front panel

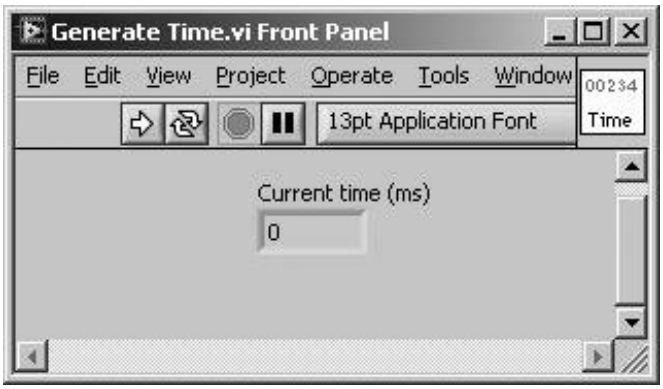
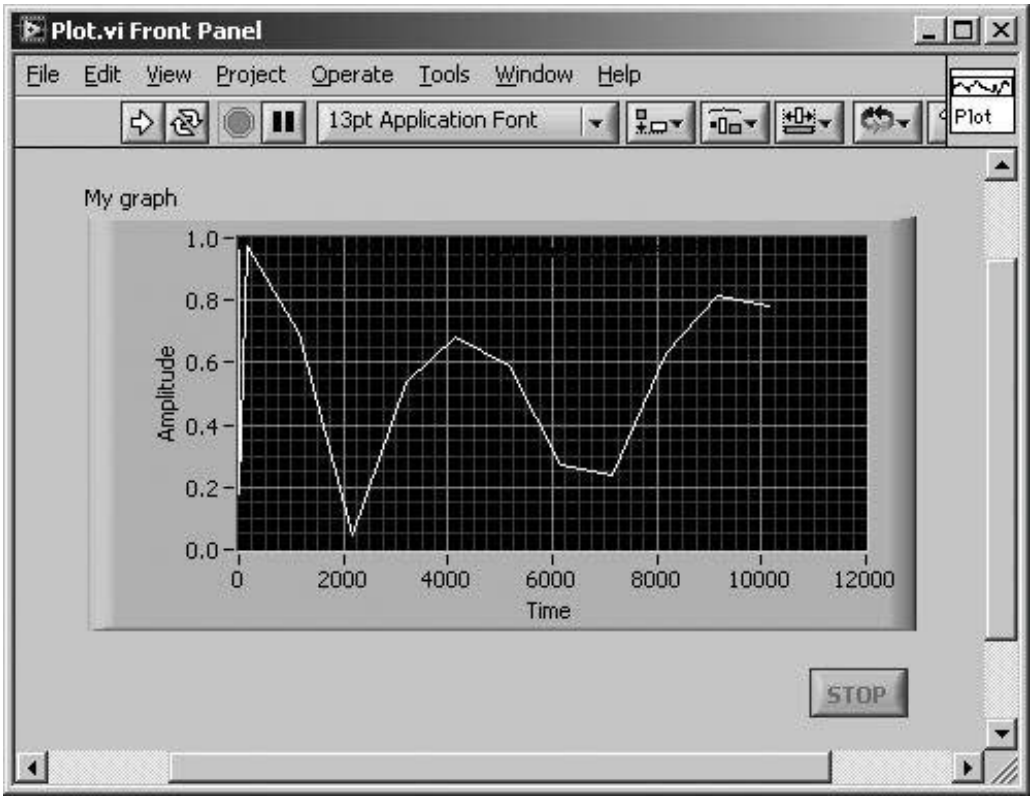


Figure 13.23. Plot.vi front panel

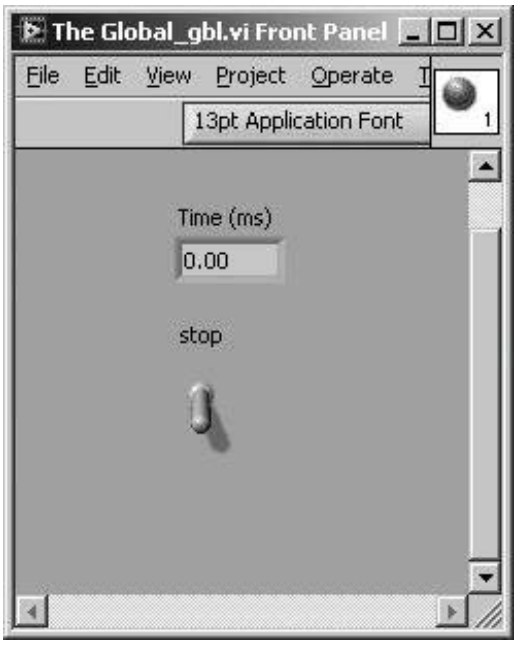


The way these two subVIs exchange data is through the use of a global. We want Plot.vi to obtain an array of time values provided by Generate Time.vi, and more importantly, we want both subVIs to be stopped by a single Boolean control.

So, first we create a global with the two necessary variables. Remember, to create a new global, select the Global Variable structure from the Programming>>Structures palette, and double-click on the "world" icon to define the global's components. In this case, we define a numeric

component `Time (ms)` and a Boolean `Stop`. The name of the global variable is The Global_gbl.vi (see Figure 13.24). (This VI may be found on the CD-ROM, in the EVERYONE\CH13 folder.)

Figure 13.24. The Global_gbl.vi front panel (a global VI)



Then we use the global's variables at the appropriate places in both subVIs (see Figures 13.25 and 13.26).

Figure 13.25. Generate Time.vi block diagram

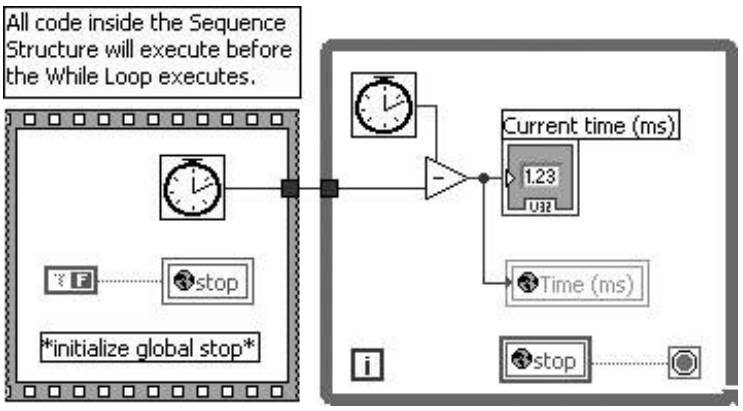
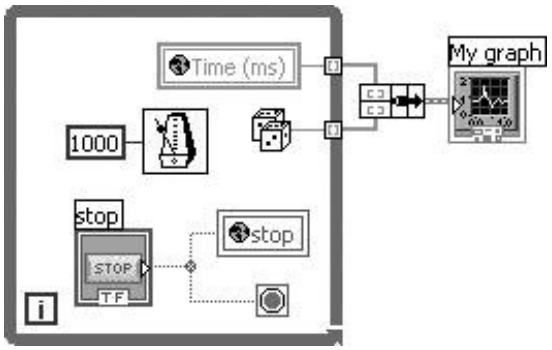


Figure 13.26. Plot.vi block diagram



Notice how the `stop` Boolean variable is used: A stop button from Plot writes to the global variable `stop`, which is in turn used to stop the While Loop in Generate Time. When the `stop` button is pressed in Plot, it will break the loop in Generate Time as well. Similarly, the time values from Generate Time are passed to the global variable `Time`, which is called by the Plot VI to build an array.

In Generate Time, the `stop` global is initialized inside of the "one-framed" sequence structure (the only good kind). Because the sequence structure outputs a data wire that is an input to the While Loop, the laws of data flow ensure that the `stop` global is initialized to FALSE before the While Loop executes and reads the `stop` global. This is a perfect example of how to avoid a race condition. If the sequence structure was not used, our code might not work correctlythe global inside the While Loop might actually be read before it is initialized outside the While Loop!

Hopefully, these two VIs have given you an example of how globals work. We possibly could have avoided using globals for what we wanted to accomplish in this program, but the simple example is good for illustrative purposes.

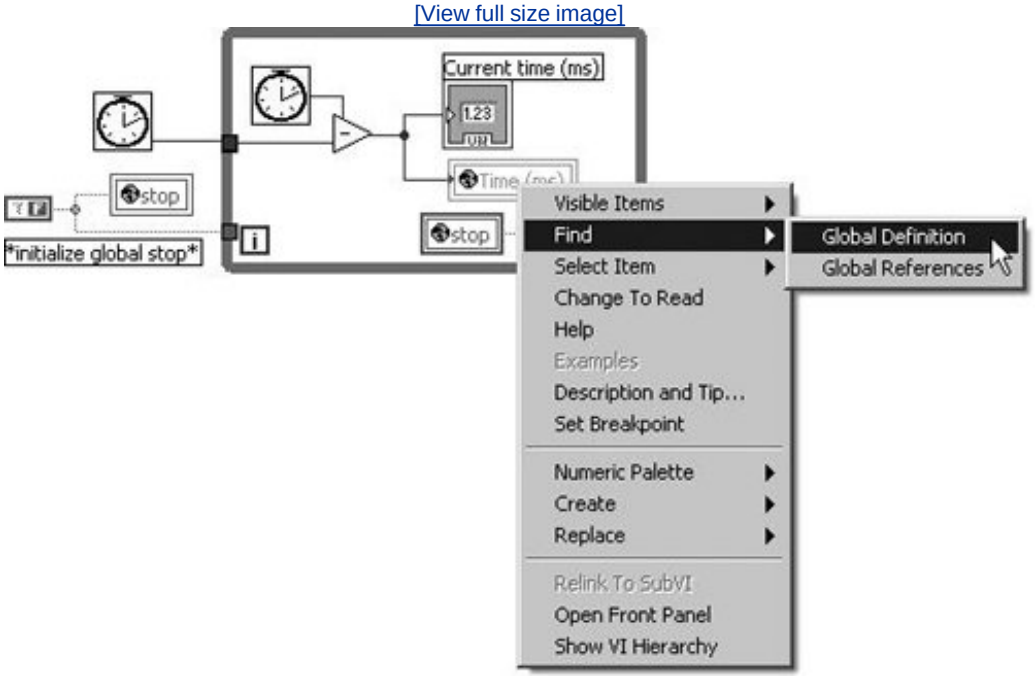
If you look at a block diagram that calls the two subVIs, you will see another problem with using globals: There are no wires anywhere! Globals obscure dataflow because we can't see how the two subVIs are related. Even when you see a global variable on the block diagram, you don't know where else it is being written to. Fortunately, LabVIEW addresses this inconvenience by allowing you to search for instances of a global.

If you pop up on a global's terminal, you can select Find>>Global Definition, which will take you to the front panel where the global is defined (see Figure 13.27). The other option is Find>>Global References, which will provide you with a list of all the VIs that contain the global. For more

information about LabVIEW's search capabilities, see Chapter 15, "Advanced LabVIEW Features."

Figure 13.27. Opening the front panel of global by selecting Find>>Global Definition from its pop-up menu

[View full size image]



Shared Variables

So you've seen how to make copies of data within a VI, using locals. You've seen how to make copies of data across multiple VIs, using globals.

Sometimes you need to go a step further and share data between two or more applications that might run on the same computer or perhaps on different computers over a network. LabVIEW has a structure called the Shared Variable (found on the Programming>>Structures palette), which is similar to a global variable, but works across multiple local and networked applications. We will learn more about this exciting feature in Chapter 16, "Connectivity in LabVIEW."

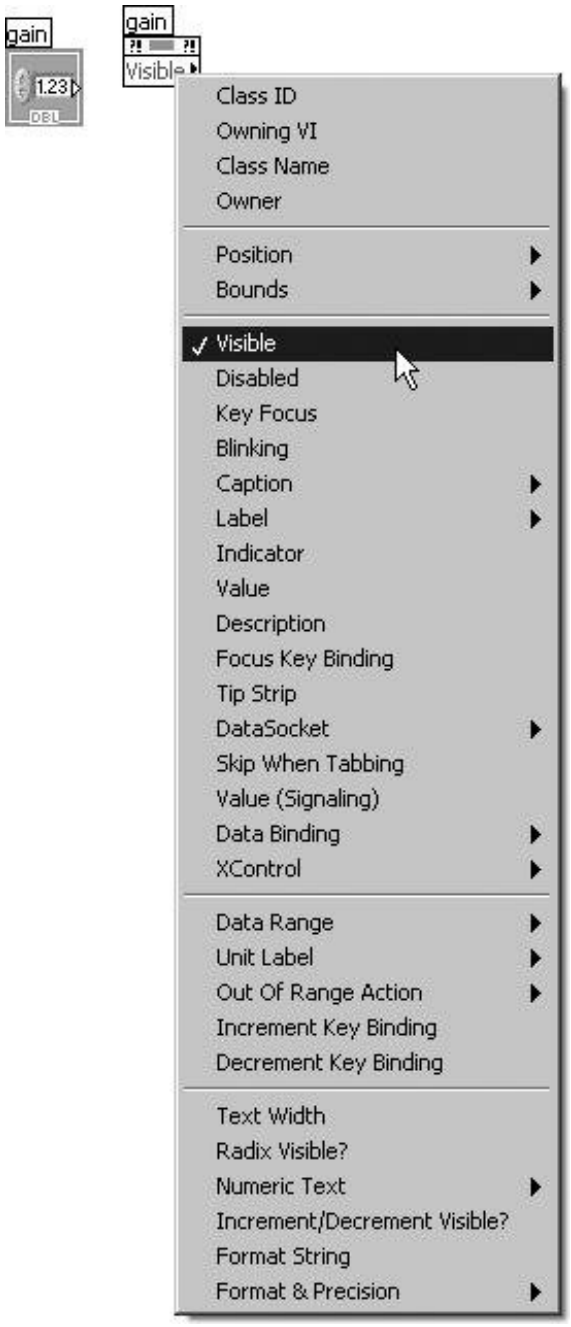
PREV NEXT

Property Nodes

With property nodes, you can start making your program more powerful and a lot more fun. Property nodes allow you to programmatically control the properties of a front panel object: things such as color, visibility, position, numeric display format, and so on. The key word here is *programmatically*that is, changing the properties of a front panel object according to an algorithm in your diagram. For example, you could change the color of a dial to go through blue, green, and red as its numerical value increases. Or you could selectively present the user with different controls, each set of them appearing or disappearing according to what buttons were pressed. You could even animate your screen by having a custom control move around to symbolize some physical process.

To create a property node, pop up on either the front panel object or its terminal and select a property from the Create>>Property Node submenu. A terminal with the same name as the variable will appear on the diagram. To see what options you can set in a control's property node, click on the node with the Operating tool or pop up on the node and choose Property (see Figure 13.28). Now you have the choice of which property or properties you wish to select. Each object has a set of *base properties* (common to all types of controls), and usually, an additional set of properties specific to that type of control.

Figure 13.28. Some of the many properties available for controls and indicators

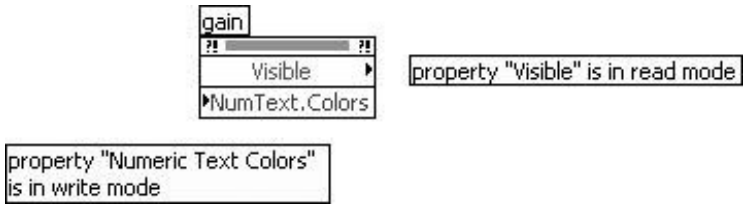


Just as with local variables, you can either read or write the property of an object (although a few properties are read-only). To change the mode of a property, pop up on it and select the Change to Write/Read option. The small arrow inside the property node's terminal tells you which mode it's in. A property node in write mode has the arrow on the left, indicating the data is flowing into the node, *writing* a new property. A property node in read mode has the arrow on the right, *reading* the current property and providing this data. The same analogy we used for locals, a control (read mode) and an indicator (write mode), holds for property nodes.

An interesting feature of property nodes is that you can use one terminal on the block diagrams for

several properties (but always affecting the same control or indicator). To add an additional property, you can use the Positioning tool to *resize* the terminal and get the number of properties you need, much in the same way multiple inputs are added to functions like Bundle, Build Array, etc. Figure 13.29 shows two properties on the same terminal for the numeric control `gain`.

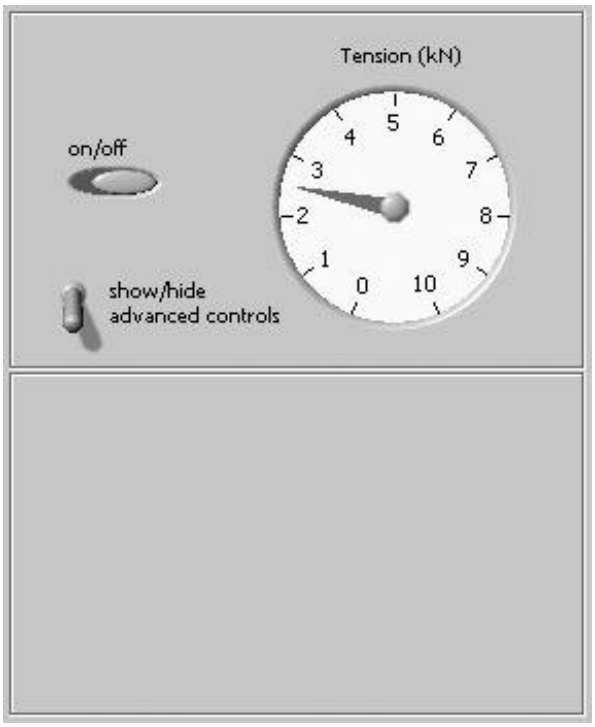
Figure 13.29. Property Node with Visible property in read mode and Numeric Text Colors in write mode



Note that the Visible property is in read mode and the NumText.Colors property is in write mode. You can configure each property from its pop-up menu by choosing Change To Write or Change To Read. You can change all properties on the node by choosing Change All To Write or Change All To Read.

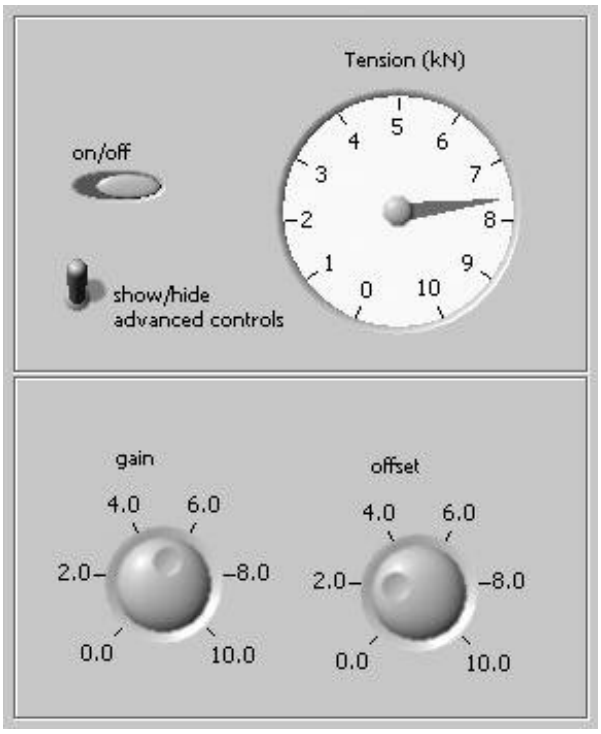
Let's look at a simple example VI, `PropertyNode Example.vi`, which may be found on the CD-ROM in the `EVERYONE\CH13` folder (see Figures 13.29 and 13.30). Suppose you wanted to have a front panel that would hide certain specialized controls except for those occasions when they were needed. In the following front panel, we see a tension gauge and a Boolean alarm switch (see Figure 13.29). We include a button that says `show/hide advanced controls...`, hinting at the possibility that if you pressed it, some really obscure and intricate options will pop up (see Figure 13.30).

Figure 13.30. PropertyNode Example.vi front panel with advanced controls hidden



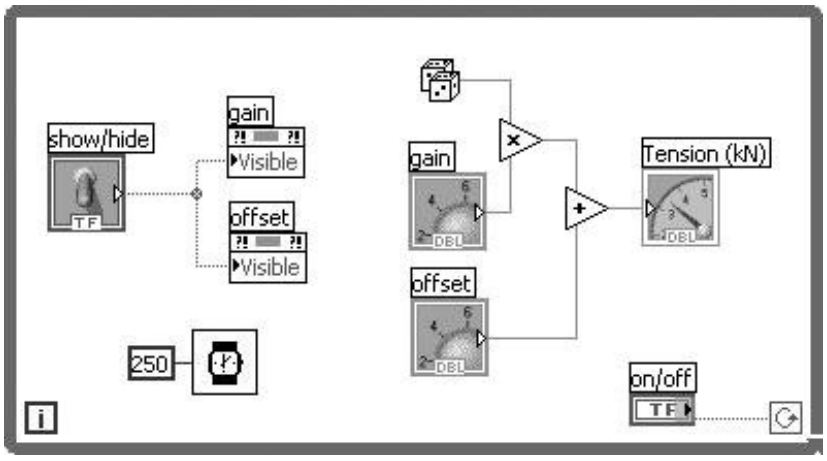
In this example, we've included two more controls, `gain` and `offset`, which are made invisible by setting their property nodes' option "Visible" to false unless the button is pressed. If the `show/hide advanced controls` . . . button is pressed, then ta-da! . . . the two knobs become visible (see Figure 13.31).

Figure 13.31. PropertyNode Example.vi front panel with advanced controls visible



The entire block diagram would be encompassed in a While Loop like the one shown next to make the button control the visibility of the two knobs and thus give the "pop-up" effect (see Figure 13.32).

Figure 13.32. PropertyNode Example.vi block diagram



You will learn about the tab control in the section, "Cool GUI Stuff: Look What I Can Do!." Once you are familiar with it, come back to this example and see if you can save some front panel real estate by putting the "advanced" controls (`gain` and `offset`) into different pages of a tab control.

Often you will want to use more than one option in an object's property node. Remember, instead of creating another property node, you can select several options at a time by enlarging the terminal with the Positioning tool (much like you enlarge cluster and array terminals). You will see each new option appear in sequence; you can later change these if you like by clicking on any item with the Operating tool.

What do some of the base options in a property node refer to?

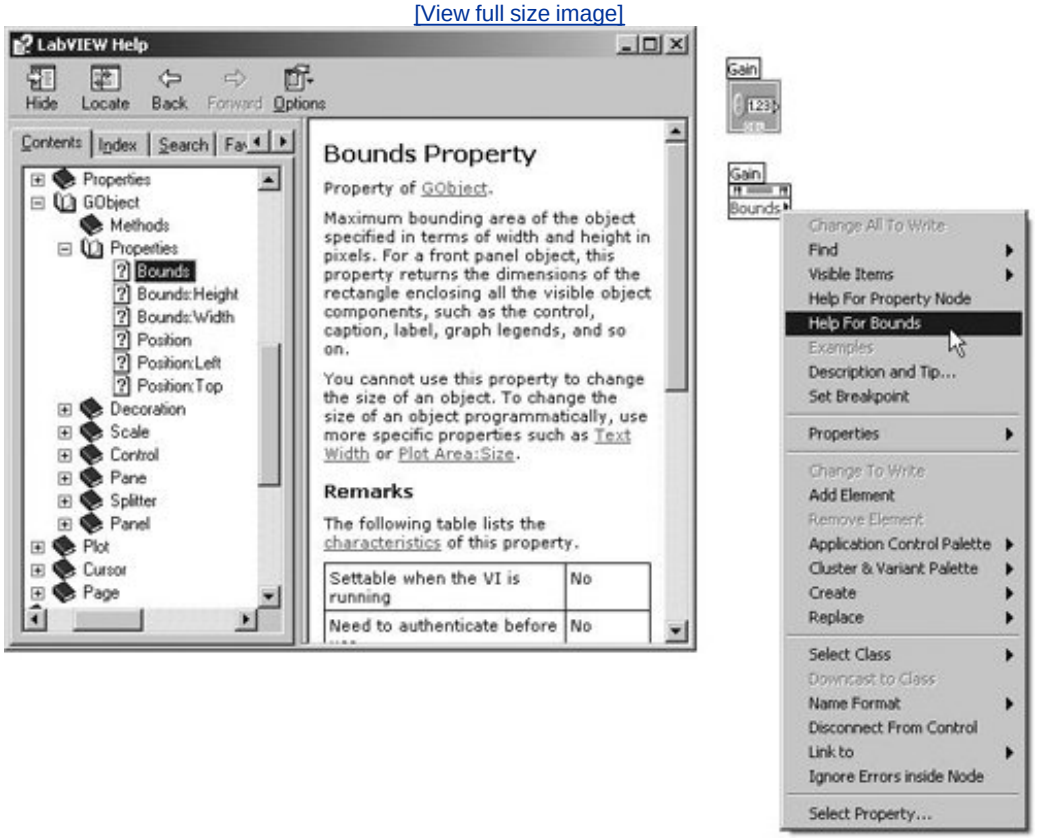
- Visible: Sets or reads the visibility status of the object. Visible when TRUE; hidden when FALSE. Use the Visible property to hide controls (or indicators) from the user. Don't try to simply set the color of a control to *transparent*transparent controls might appear to be invisible, but they are not hidden and the user may still (unknowingly) interact with them.
- Disabled: Sets or reads the user access status of a control. A value of 0 enables the control so the user can access it; a value of 1 disables the control without any visible indication; and a value of 2 disables the control and "grays it out."
- Key Focus: When true, the control is the currently selected key focus, which means that the cursor is active in this field. Key focus is generally changed by tabbing through fields. Useful for building a mouseless application. See Chapter 15 for more information on key focus.
- Position: A cluster of two numbers that respectively define the top and left pixel position of the front panel object.
- Bounds: A cluster of two numbers that respectively define the height and width in pixels of the entire front panel object.
- Blinking: When true, the front panel object blinks.
- Format and Precision: Sets or reads the format and precision properties for numeric controls and indicators. The input cluster contains two integers: one for format and one for precision. These are the same properties you can set from the pop-up menu of the numeric object.
- Color: Depending on the type of object, you may have several color options. The input is one of those neat color boxes that sets the color of the text, background, etc., depending on the object.
- Tip Strip: This is the text that appears when the user holds the mouse still over the control.

- You'll also notice every object has properties called ClassID, ClassName, Owning VI, Owner. Don't worry about them at this point. They are for very advanced and relatively obscure LabVIEW programming scenarios.

The Help window really is helpful when using property nodes. If you move the cursor onto the terminal of a property node, the Help window will show you what the property means, and what kind of data it expects. Also, you can pop up on a property node and select Help to open the LabVIEW help file topic for the property that you popped up on, as shown in Figure 13.33.

Figure 13.33. Showing the LabVIEW help for a specific property from its pop-up menu

[View full size image]



If you are unsure about the data type to wire to a property (in write mode), you can pop up on the property node terminal and choose Create Constant to create a constant of the correct data typethis is an especially handy time-saver when the input is a cluster.

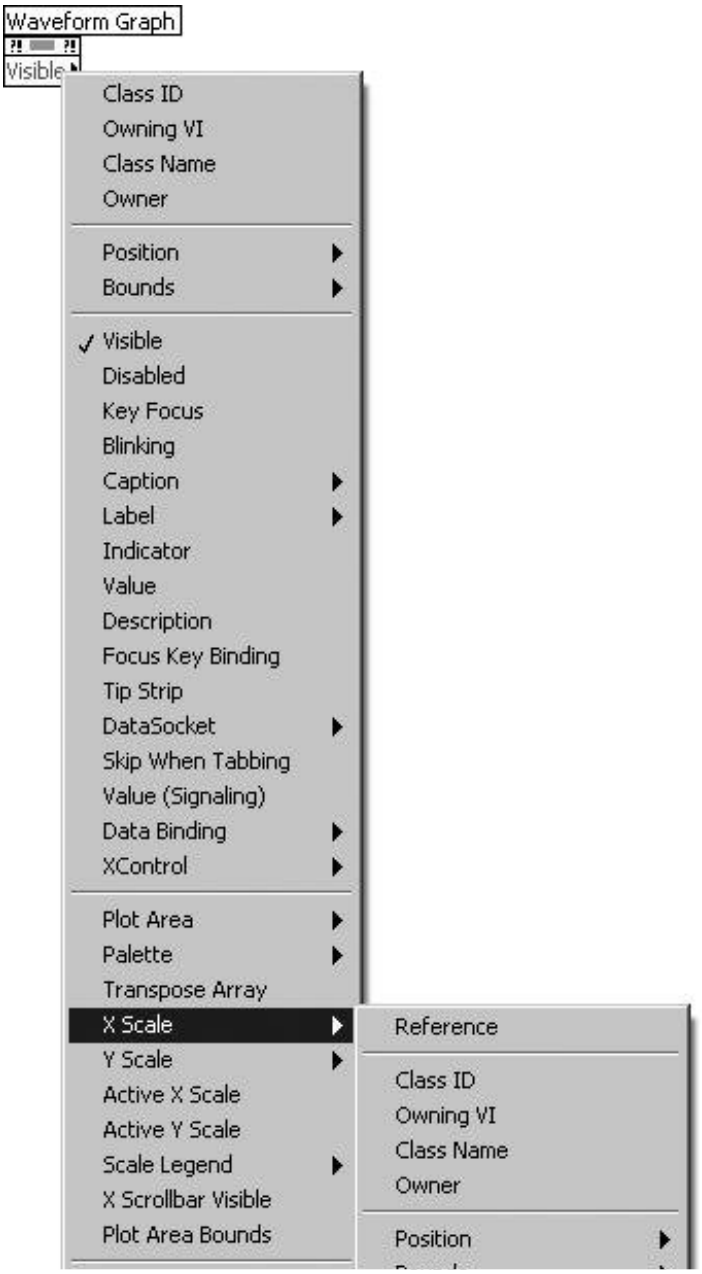
Almost all controls or indicators have the base properties. Most of them have many more, especially tables and graphs (which can have over 100 properties!). We won't even begin to go into most of these properties, partly because you may never care about many of them and you can always look up the details in the manuals. The best way to learn about property nodes is to create some to go with your application and begin to play around with them. You'll find that property nodes are very

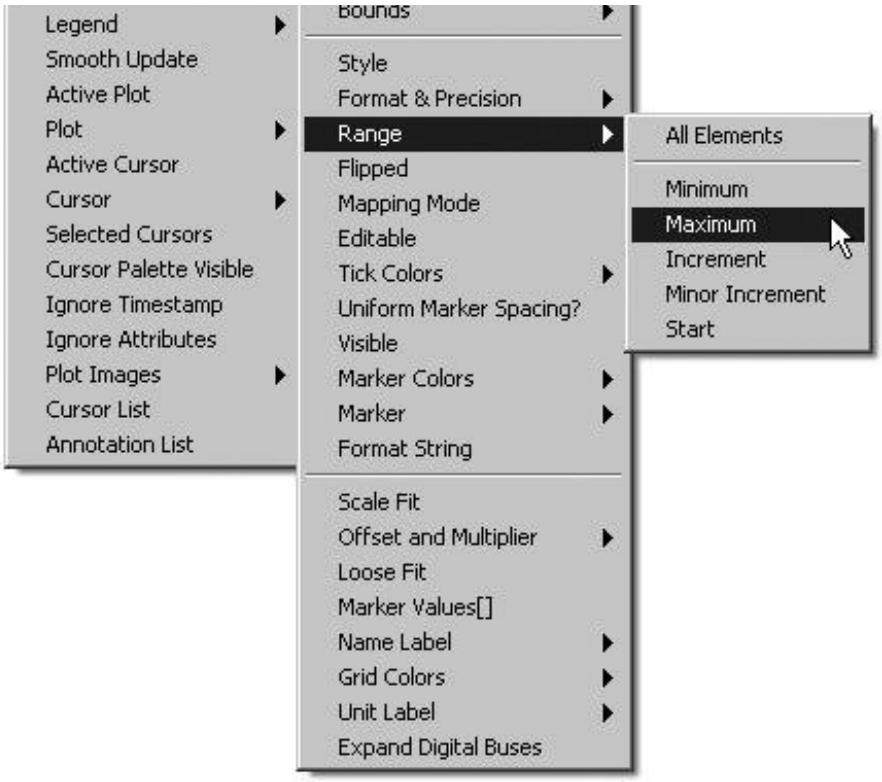
handy for making your program more dynamic, flexible, and user-friendly (always good for impressing your non-technical manager).

Another Example

Graphs and charts have zillions of options in their property nodes, as you can see by popping up on the terminal of a chart's property node (see Figure 13.34).

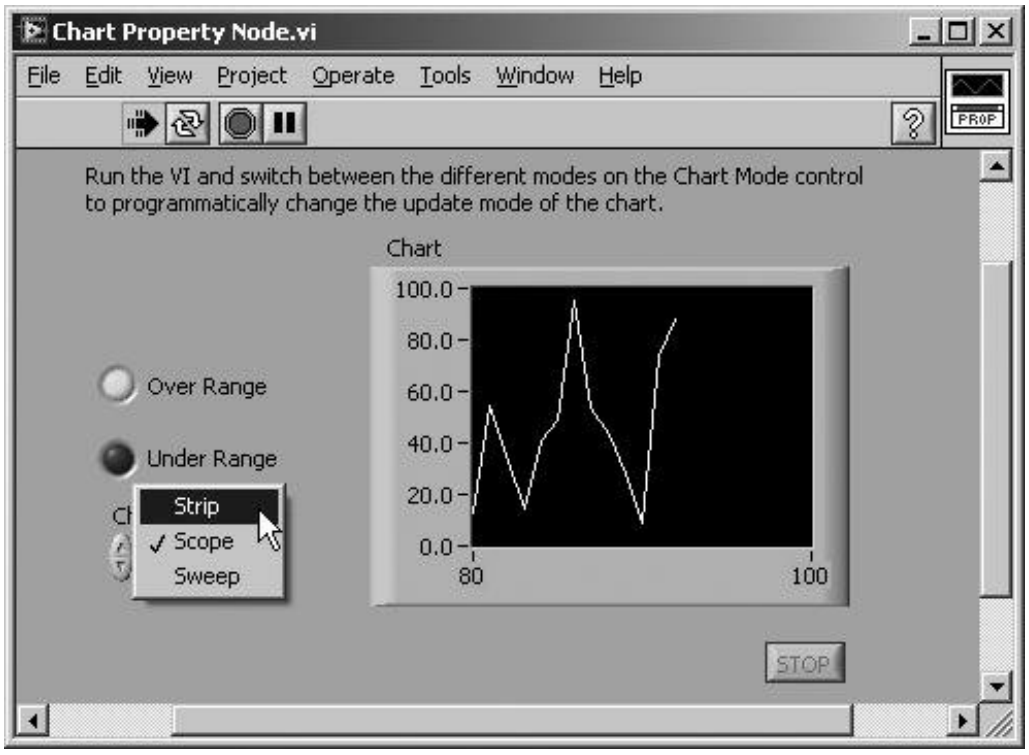
Figure 13.34. The zillions of properties of charts and graphs





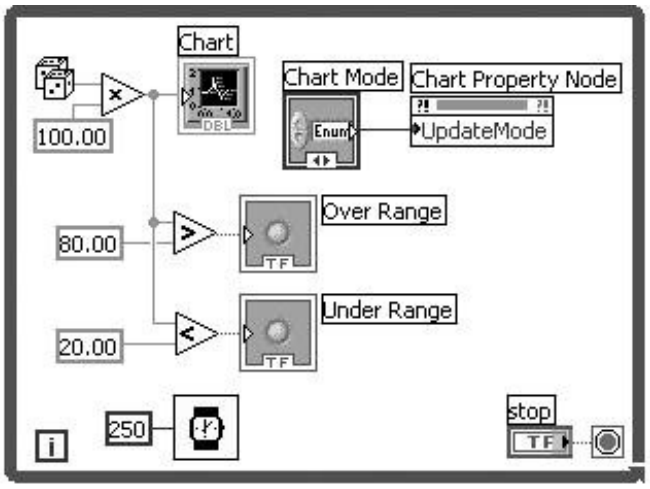
This next example, which can be found in the property node examples in the full version of LabVIEW, shows just one of the many aspects of a graph you can control programmatically (see Figure 13.35). Chart Property Node.vi lets you select one of three display types for the chart: Strip, Scope, and Sweep (if you need to, review Chapter 8, "LabVIEW's Exciting Visual Displays: Charts and Graphs," to see what they do).

Figure 13.35. Chart Property Node.vi front panel



You can select the chart mode and watch it change even while the VI is running. The way this is done is through the Update Mode option on a chart's property node (see Figure 13.36).

Figure 13.36. Chart Property Node.vi block diagram



Inside the loop is a property node that alters the update mode of the chart according to the setting specified by the Chart Mode ring control of the front panel.

Activity 13-4: Using Property Nodes with Charts

Write a VI that graphs three channels of data (either real-time data through a DAQ board, or random data). Let the user turn on or off any of the plots with three buttons. Because this is a chart and not a graph, the data will accumulate along with the old data, even if the VI is closed and re-opened. Add a "CLEAR" button that clears the chart (see Figure 13.37).

Figure 13.37. Front panel of the VI you will create during this activity

When using a multiplot graph or chart, you can only affect the properties of one plot at a time. The plots are numbered 0, 1, . . ., n for property node purposes. A specific property, called Active Plot, is used to select the plot for the properties you are modifying or reading. In this exercise, you will need to build some case statements like the ones shown in Figures 13.38 and 13.39.

Figure 13.38. Part of the block diagram you will create, which shows plot 0

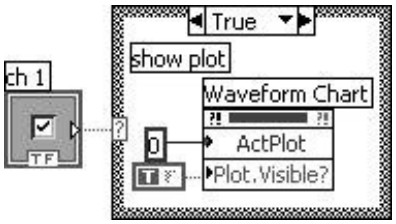
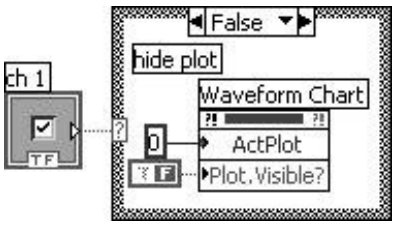


Figure 13.39. Part of the block diagram you will create, which hides plot 0



The Boolean text of the checkbox (or any Boolean, for that matter) is clickable, meaning that when you click on the text, it is the same as clicking on the Booleanit will change states. When you place a checkbox (found on the System palette), it will appear with both its label and Boolean text visible. Hide the label (right-click the checkbox and unselect Label from the Visible Items submenu) and change the Boolean text to anything you wishin our case "Channel 1 Visible" (etc). Test whether clicking the Boolean text causes the checkbox to change values.

To clear a chart, use the History Data property of the chart. Then wire an empty array to this property node.

Save your VI as `Property Nodes-Graph.vi`.

PREV NEXT

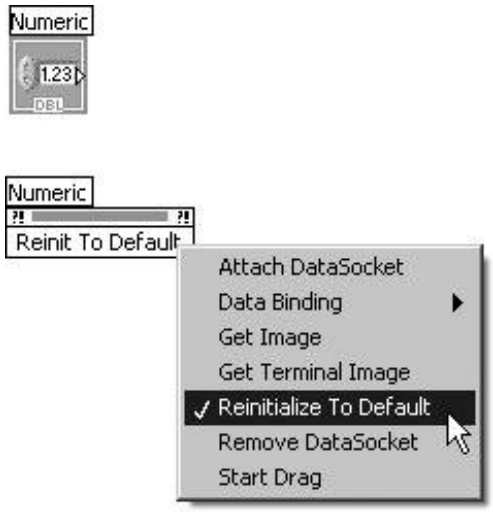
Invoke Nodes

Invoke nodes are very similar to property nodes. Calling an invoke node runs a single method on the front panel object and sometimes requires inputs, (also known as "arguments"). In a way, calling an invoke node is similar to calling a subVI or calling a function or method in other languages.

The difference between a property node and an invoke node is that calling an invoke node, "executes something"it doesn't just change the value of a property. You can think of invoke nodes as functions that you execute; whereas a property node is a property, or state, that can be read from or written to.

To create an invoke node, pop up on either the front panel object or its terminal, and select one of the control's methods from the Create>>Invoke node>> short-cut menu. A terminal with the same name as the variable will appear on the diagram. The method name will appear as the first item in the invoke node, and all the arguments will appear as items beneath the method name. You can change the method being called by the invoke node with the Operating tool or pop up on the node and choose Methods>>. Now you have the choice of which method you wish to select. Each object has a set of *base methods*, and usually, an additional set of methods specific to that object. One base method that is available for all controls is Reinitialize to Default (see Figure 13.40). It doesn't have any arguments, and calling it will do just what it saysreinitialize the control to its default value.

Figure 13.40. Configuring a control's invoke node to call the Reinitialize to Default method



Activity 13-5: Using Invoke Nodes to Export an Image from a Graph

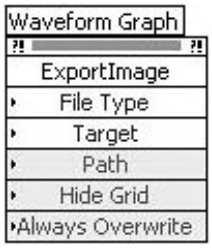
In this activity, you'll see how to use an invoke node specific to graphs, which allows you to export the graph to an image file.

We'll use the VI you created in Chapter 8, Graph Sine Array.vi, so you should open that (or use the one on CD-ROM, in the `EVERYONE\CH08` folder).

1. Open Graph Sine Array.vi.

2. From the block diagram, right-click on Waveform Graph and select Create>>Invoke Node>> ExportImage. You should see an invoke node like the one in Figure 13.41.

Figure 13.41. Invoke node configured to call the Waveform Graph's Export Image method



3. The Export Image method creates an image of the graph it's associated with, and sends this image to the clipboard or to a file. The following inputs are required:

   a. *FileType* can be BMP, EPS, EMF, or PICT.

   b. *Target* can be Clipboard or File.

   The following inputs are optional:

   c. *Path* is the path to the file to be saved.

   d. *HideGrid* is a Boolean, which if TRUE, hides the graph's grid on the image.

   e. *AlwaysOverwrite* is a Boolean, which if TRUE, overwrites the image file even if one with the same path and filename exists.

4. To use this node, you can pop up on each of its inputs and select the appropriate value. Set FileType to "BMP" and Target to "File."

5. For the path, it will be best if we allow the user to choose the path with a dialog. Here's an easy

way to program a prompt for the user to enter a filename: Use the Express VI File Dialog (shown in Figure 13.42), which may be found on the Express>>Input palette.

Figure 13.42. File Dialog express VI

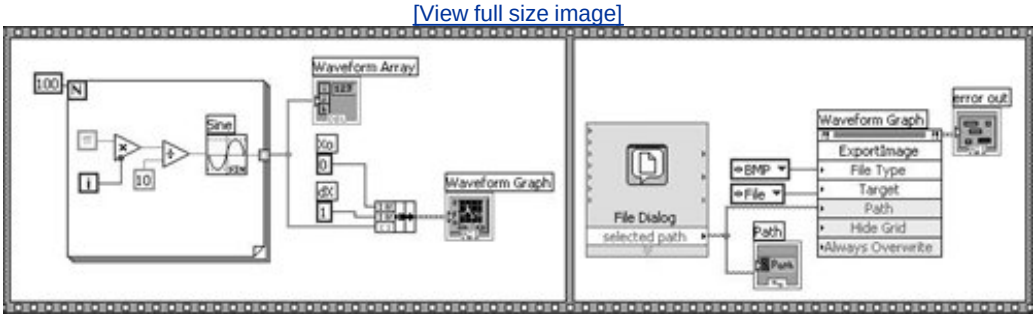
When you drop the File Dialog Express VI onto the block diagram, you will be asked to configure some options for this dialog. Select "Files only" and "New or existing."

6. Finally, use a sequence structure to make sure that things happen in the right order, and wire the output of File Dialog to the "path" input in ExportImage.

   Your VI's block diagram should look like Figure 13.43.

Figure 13.43. Block diagram of the VI you will create during this activity

[View full size image]



7. Save your VI as `Invoke Node Activity.vi` in the `MYWORK` directory.

When you run it, you should be able to choose a filename to save the bitmap under. Open the bitmap

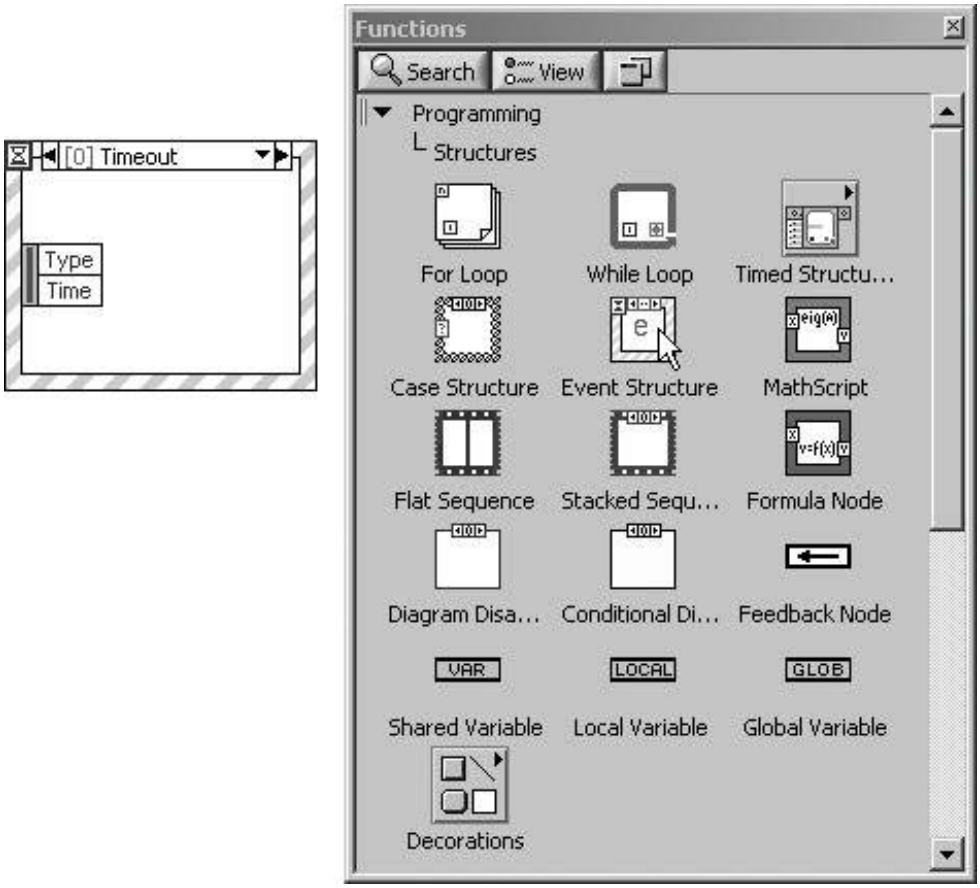
file this VI creates and you'll see an image created from the waveform graph.

PREV NEXT

Event-Driven Programming: The Event Structure

The Event Structure (found on the Programming>>Structures palette) is an extremely powerful programming tool (see Figure 13.44). It allows you to write highly efficient code that *waits for events to happen*, rather than inefficient code that periodically checks whether events have happened.

Figure 13.44. Placing an Event Structure onto the block diagram from the Programming>>Structures palette



So what is an event anyway? An event can be almost anything that "happens" in LabVIEW, for example:

- You press a front panel Boolean.
- The value of a numeric control changes.
- The mouse cursor enters the VI window.
- A key is pressed.

As you can see, events usually refer to GUI events, but can also refer to front panel controls changing their value. You can even define your own custom events.

Without the Event Structure, in order to detect if a user pressed a STOP button, for example, you have to "poll" its value periodically in a While Loop (like you've done in several of the activities). With the Event Structure, you don't have to "poll" because the VI will "know" when that event occurs.

The Event Structure looks similar to a Case Structure, in that it has multiple "case" frames. Each case of an Event Structure can be registered to handle one or more events. When an Event Structure executes, it will wait until an event has occurred and then execute exactly one case framethe case that is configured to handle the event that has occurred.

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

The Timeout Event

When we first place an Event Structure on the block diagram, it will be preconfigured with only one event case that is set to handle the Timeout event. The Timeout event is a special event that gets triggered if no other events (that the Event Structure is configured to handle) happen before the timeout value occurs.

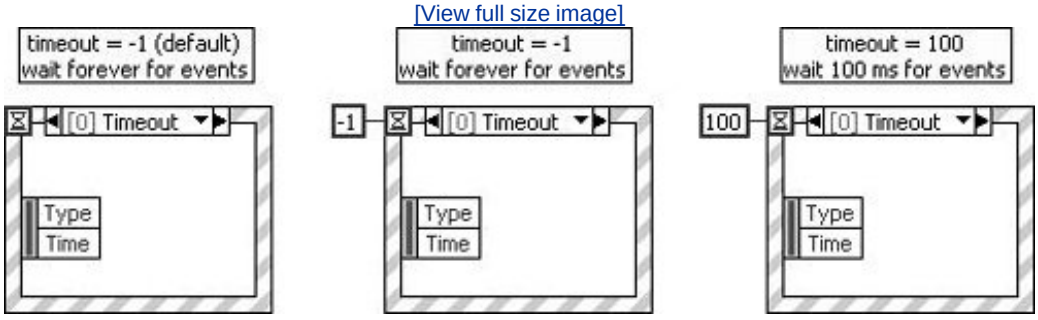
Timeout Terminal

You can specify the Event Structure's timeout value by passing it to the Timeout Terminal, in the upper-left corner of the Event Structure. The default timeout value (if unwired) is 1, which means "never time out" or "wait indefinitely."

Figure 13.45 shows three different timeout configurations of the Event Structure. Note that the first two (left and middle) are equivalent.

Figure 13.45. Event Structures with different timeout values

[View full size image]



If you set a timeout value of -1 and no events ever occur, then your VI will appear to be non-responsive. This sort of situation is similar to an infinite loop, which exhibits the same non-responsive behavior. This situation can be avoided by either setting a reasonable timeout value, or making sure that the Event Structure is also registered for other events (besides just the Timeout event).

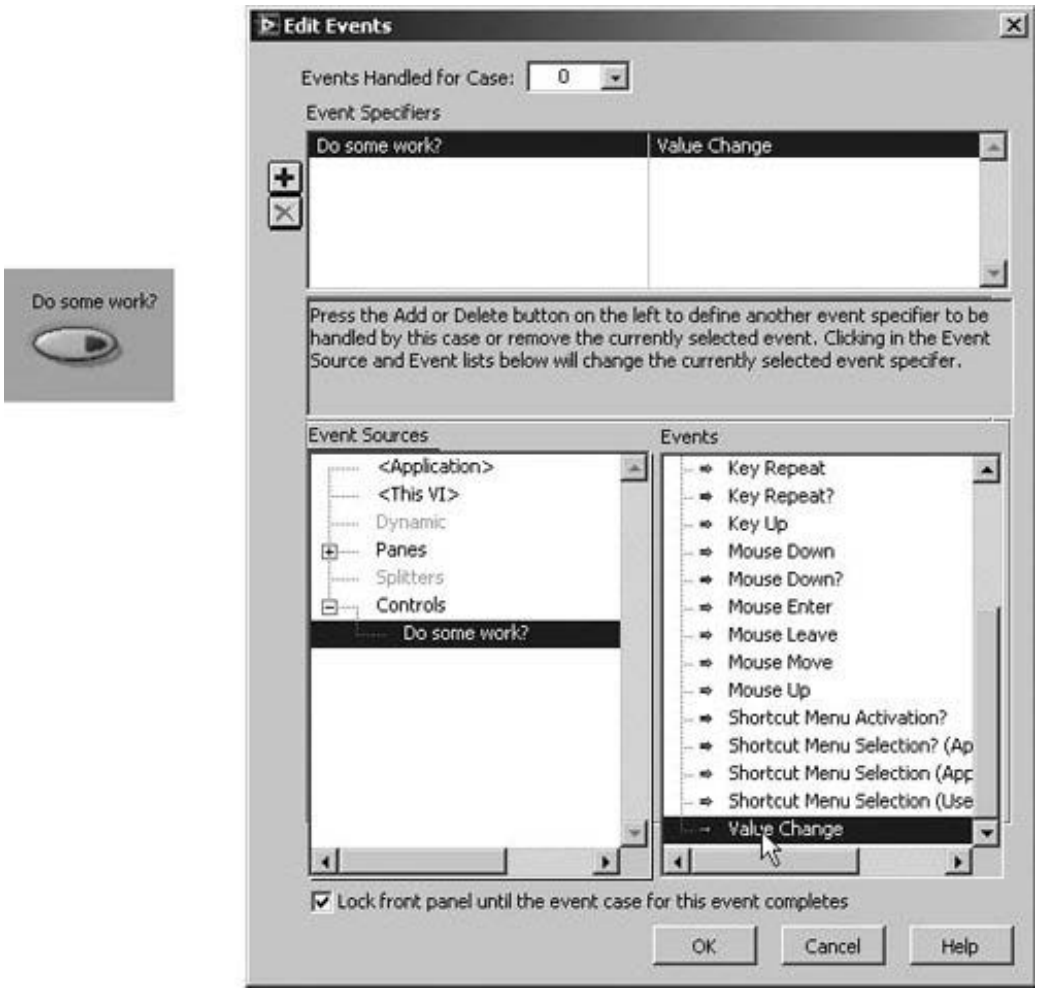
Editing Events Handled by Event Structure Cases

To register an Event Structure to handle events of front panel controls, pop up on the frame of the Event Structure and select Add Event Case . . . . This will open the Edit Events dialog.

For example, if you have Boolean control on the front panel called "Do some work?," you could create a *Value Change* event for it by popping up on the Add Event Case, selecting that control, and selecting the Value Change event. This is shown in Figure 13.46.

Figure 13.46. Edit Events dialog, which is used for configuring event cases of an Event Structure, alongside the front panel control that is configuring as an event source

[View full size image]



This dialog has the following components:

- The `Events Handled for Case` ring lists the number and name of all cases for that Event Structure. You can select a case from this pull-down menu and edit the events for that case. When you switch to a different case, the Event Structure on the block diagram updates to display the case you selected.
- The `Event Specifiers` listbox lists the event source (Application, VI, Dynamic, or Control) and event name of all the events the current case of the Event Structure handles.
- The `Insert Event` button is used to add a new event to be handled by the current case.

  Insert Event Button
- The `Delete Event` button is used to delete the event selected in the Event Specifiers list.

Delete Event Button

- The `Event Sources` tree control lists the event sources, sorted by class, you can configure to generate events.
- The `Events` listbox lists the events available for the event source you select in the Event Sources section of the dialog box.
- The `Lock front panel until the event case for this event completes checkbox` locks the front panel when this event occurs. LabVIEW keeps the front panel locked until all Event Structures finish handling this event. You can change this setting for notify events but not for filter events, which *always* have this setting enabled. (You'll learn the difference between notify and filter events later.)

From the Event Structure's pop-up menu, you can also select Edit Events Handled by This Case . . . to modify an existing case, Duplicate Event Case . . . to copy a case, or Delete This Event Case to delete an existing case.

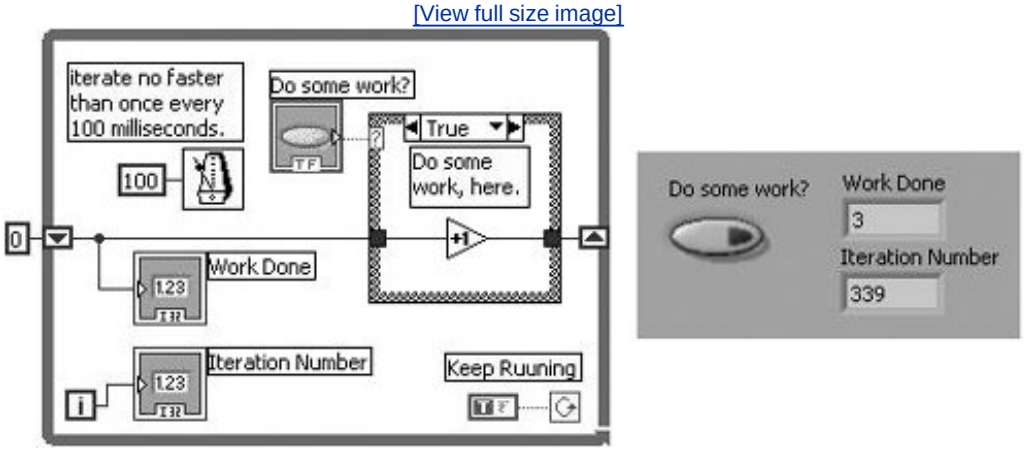
Now that we have talked about how to edit the Event Structure to handle events, let's look at an example of how to use the Event Structure.

Using the Event Structure

Without using an Event Structure, we can detect events such as pressing a Boolean control by periodically testing the value of the control. This technique is called *polling*. Figure 13.47 shows an example of how to poll the value of Boolean control in order to do work once the user presses it. Note that the Boolean control has been configured with the *Latch When Released* mechanical action so that it "bounces back" to FALSE immediately after the True value is read by LabVIEW.

Figure 13.47. Capturing front panel events by polling, without the use of an Event Structure

[View full size image]



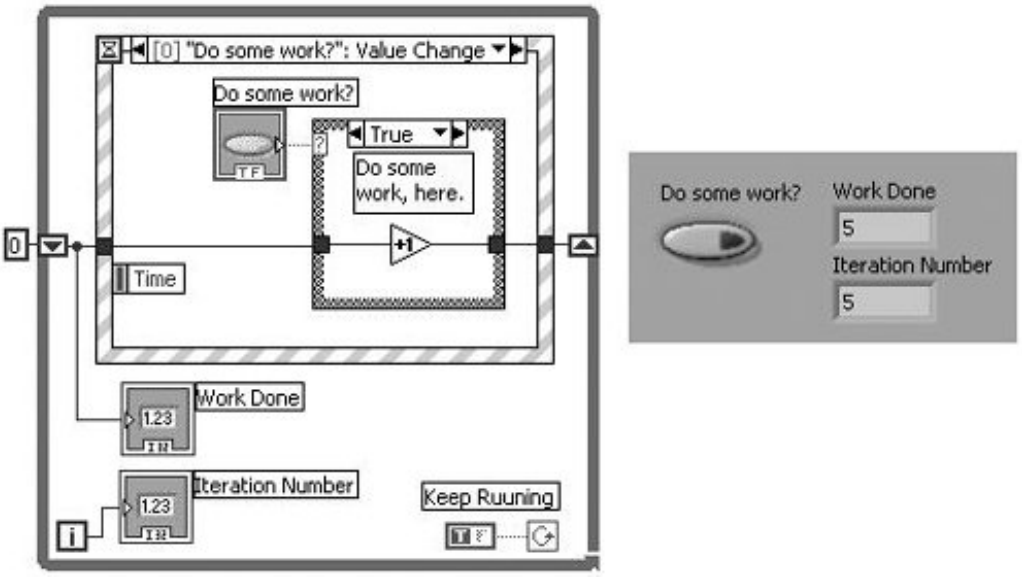
In this example, we can see that the user has pressed the `Do some work?` button three times, but the While Loop has iterated a total of 339 times! This loop may run for days, iterating millions of times before the button is actually pushed. Of course, this is no problem for most computers, because we have specified that the While Loop should not execute more than once per 100 millisecondsthis leaves a lot of computational resource available for other tasks. (However, this also means that there can be up to 100 milliseconds between when the user presses the button and when the value is read (and responded to) inside the polling loop.)

But, what if there were a way to put the While Loop to sleep (so that it does not iterate) *until* the button was pushed? That would mean that our While Loop would only iterate once for every time that our button was pushed. Guess what? That's exactly what the Event Structure does!

Figure 13.48 shows an Event Structure that waits on the Value Change event of our `Do some work?` button. (The Value Change event is generated when the user changes the value of a control.) As we can see in Figure 13.48, the user has pushed the `Do some work?` button five times, and the While Loop has only iterated five timesone time for every push of the button! *Additionally, there is no detectable delay between when the button is pushed and when the value is read inside the Event Structurethis is much more efficient and responsive than the polling technique*.

Figure 13.48. Capturing front panel events using an Event Structure, without having to poll

[View full size image]



Placing a single Event Structure inside a While Loop is the most common (and probably best) way to use the Event Structure. We can see from direct comparison between the polling example in Figure 13.47 and the Event Structure example in Figure 13.48 that our code does not look dramatically different. All we have done is (1) wrapped our `Do some work?` terminal and Case Structure in an Event Structure and (2) removed the Wait Until Next ms Multiple function because the Event Structure will wait indefinitely (Timeout = -1) for a Value Change event to occur.

An Event Structure will capture events that occur at any time while the VI is running, not just while the Event Structure is waiting. When the VI first starts running, the Event Structure creates an event queue, behind the scenes, for buffering events. As events occur, they are put into this event queue. When the Event Structure does execute, it dequeues the oldest event from the queue (First In First Out [FIFO]). If no events are in the queue, it will wait until the next event occurs. So, the bottom line is that the Event Structure will never "miss" events that happen while it is handling events, or after it handles events (before the While Loop reiterates and calls the Event Structure again). Those events will remain in the event queue until the Event Structure has handled them.

There are a few important guidelines for using the Event Structure:

- *Always put an Event Structure inside a While Loop. The Event Structure only handles one event at a time, so you need to keep executing the Event Structure, in order to keep handling events.*
- *Do not put more than one Event Structure in your VI. If you want to handle several different events in your VI, then create multiple case frames (for handling these events) inside a single Event Structure.*

Event Smorgasbord

The most common events you'll probably use are the Timeout and the Value Change events. However, keep in mind there are a myriad of other events that can be used to do things that you could never dream of easily doing in LabVIEW, were it not for the Event Structure. For example:

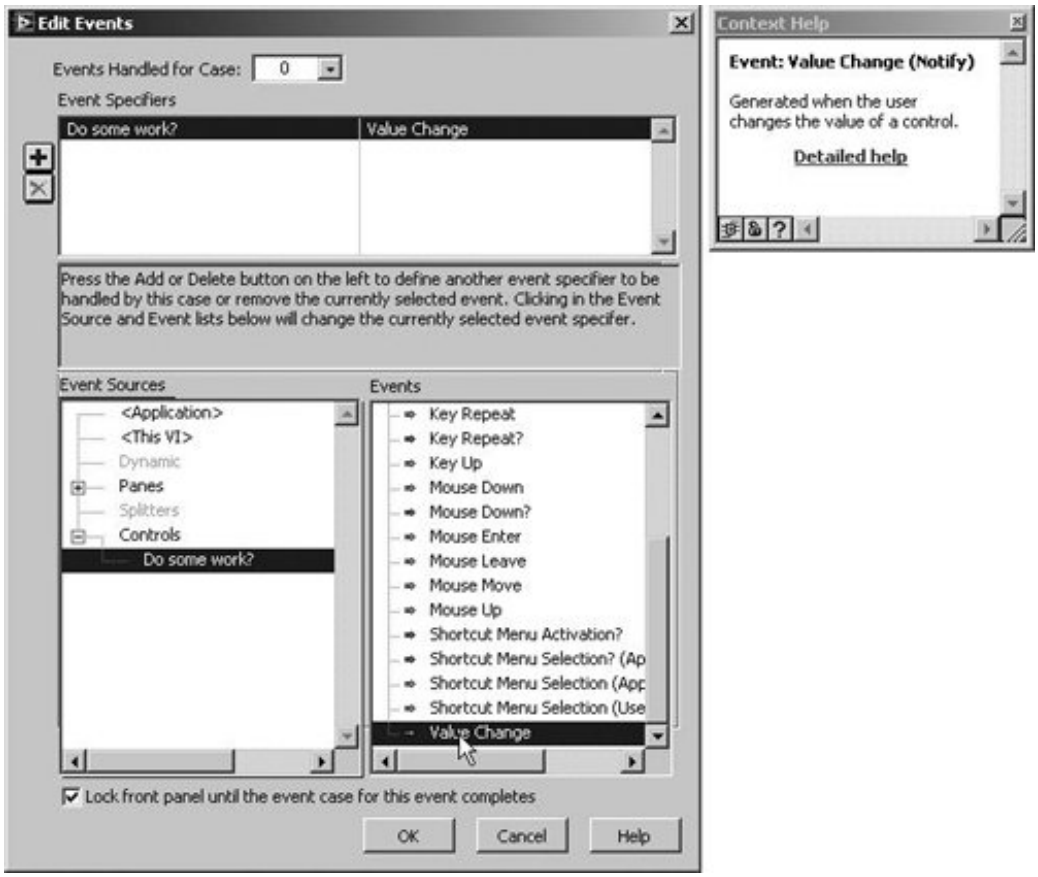
- The mouse entering or leaving a VI's front panel (or specific controls on the front panel).
- The mouse moving over a control.
- The mouse button being pressed or released (even telling you which mouse button was clicked and whether it was a double-click).
- The VI's front panel being resized, or something being dragged onto a control (as in drag and drop, which we will learn about shortly).

We could go on for days describing all of the events available in LabVIEW; however, we will leave it up to you to explore the events available in the next activity.

Use the Edit Events dialog in conjunction with the Context Help window to explore the various events available in your VI and its front panel controls. When the mouse cursor is moved over an event name in the Events list of the Edit Events dialog, the context help window will display a description of that event (see Figure 13.49).

Figure 13.49. The Context Help window showing useful documentation for an event in the Edit Events dialog, as the mouse is hovered over the event name

[View full size image]



Activity 13-6: Playing with Events

In this activity, you'll use a simple LabVIEW example to gain insight into how events work and what triggers them.

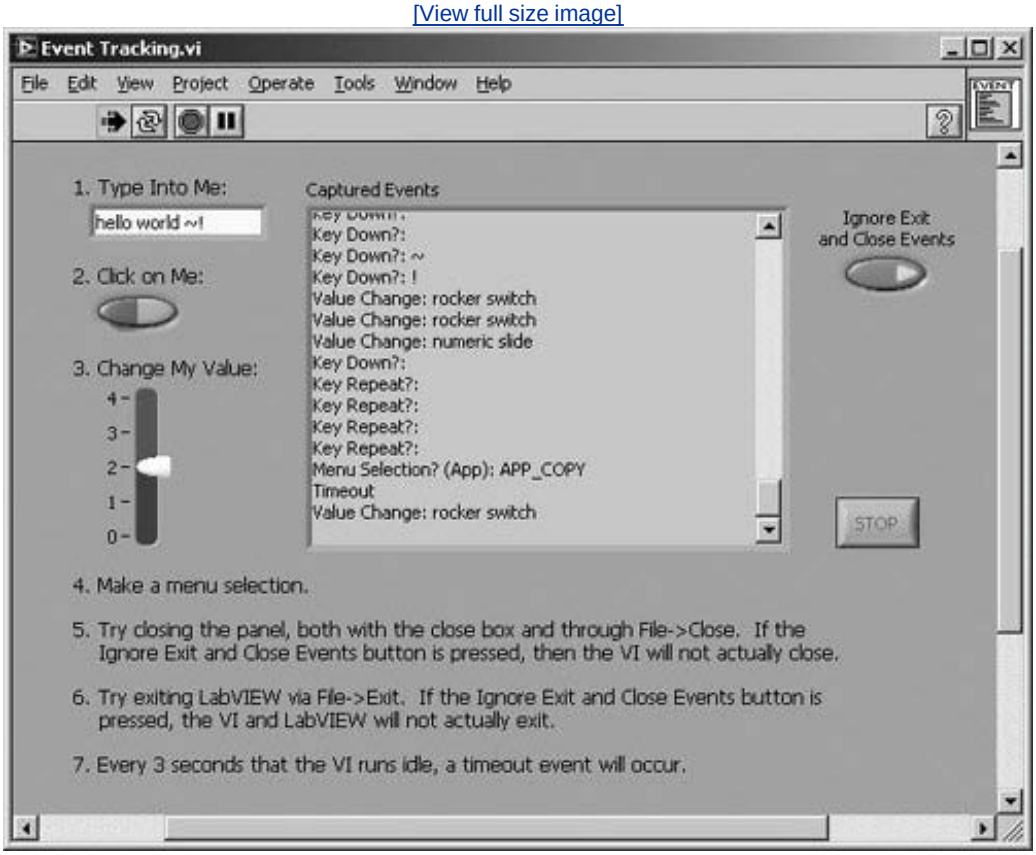
1. Open the built-in LabVIEW example Event Tracking.vi, which you can find in the LabVIEW examples folder at `examples\general\uievents.llb`.

2. Run the VI and play with the front panel controls by typing into the textbox, clicking on the Boolean, and moving the slider.

3. You'll see a textbox that gives you a real-time log of the events the VI is capturing (see Figure 13.50).

Figure 13.50. Event Tracking.vi

[View full size image]

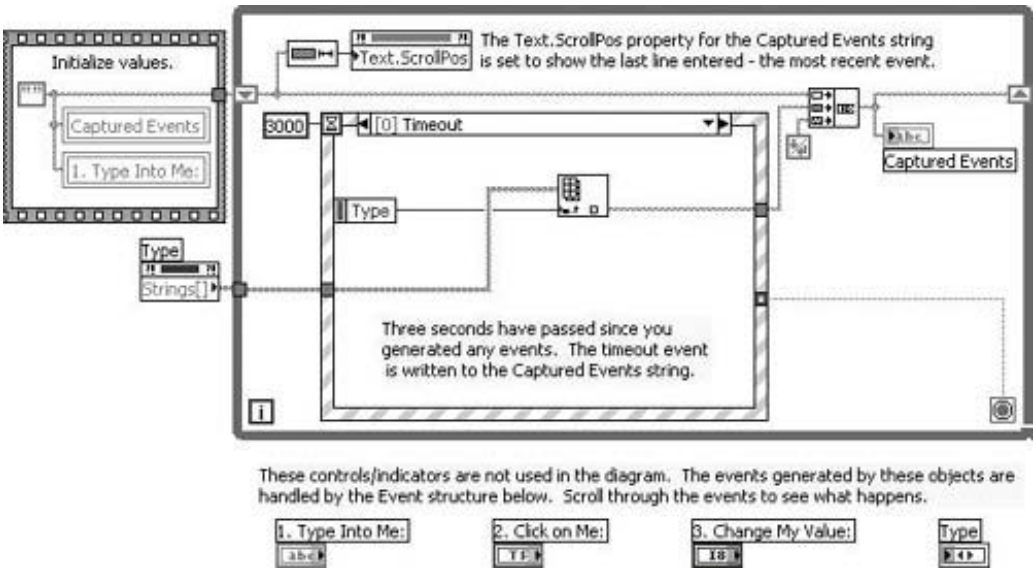


4. Open the block diagram and study the code. Notice how eight events have been defined in the Event Structure: Timeout, "Type Into Me: Key Down?," "Panel Close," and so on. Notice how inside each Event Structure case, the code updates the "Captured Events" textbox.

5. See if you can trigger each of the eight events from the front panel (see Figure 13.51)!

Figure 13.51. Event Tracking.vi block diagram

[View full size image]



Stopping While Loops That Contain Event Structures

Stopping a While Loop that contains an Event Structure can be done with a stop button (Boolean control) wired to the conditional terminal of the while loop, just as we might do for a While Loop that is polling control values. However, there is one major difference: *We must place the stop button inside the Event Structure, in an event case that is configured to handle the Value Change event for the stop button.* If the stop button is outside the Event Structure, the Event Structure will continue to wait, even after we have pressed the stop button (see Figures 13.52 and 13.53).

Figure 13.52. The *correct* way to stop a While Loop with Event Structure

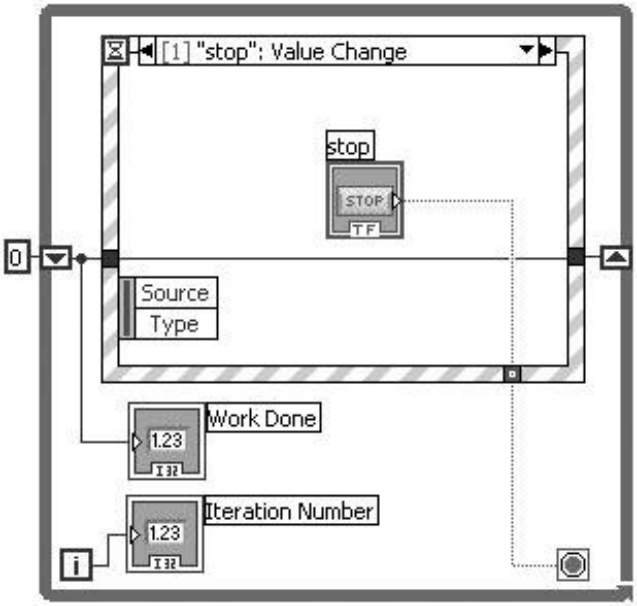
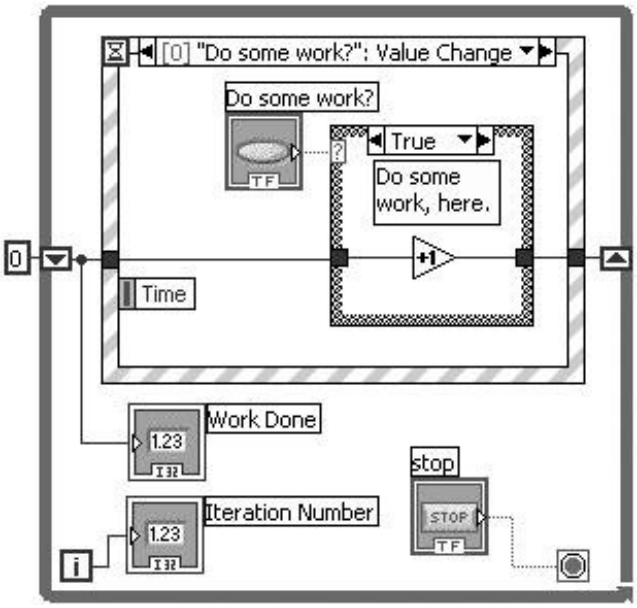


Figure 13.53. The *incorrect* way to stop a While Loop with Event Structure



Use Default if Unwired

Stop if True

One thing to pay attention to when wiring the `stop` button through the frame of the Event Structure is that the output terminal is configured to "Use Default if Unwired" (this setting is accessible from the pop-up menu of the output terminal). This means that if some other case of our Event Structure executes, a value of False (the default value of the Boolean data type) will flow out of this terminal. Because the conditional terminal of the While Loop is set to `Stop if True`, the While Loop will not stop running, unless the stop button's `Value Change` event occurs and the value of the `stop` button is True.

Whenever an output terminal is created by wiring through the frame of an Event Structure, the output terminal will be automatically configured to `Use Default if Unwired` (this is the

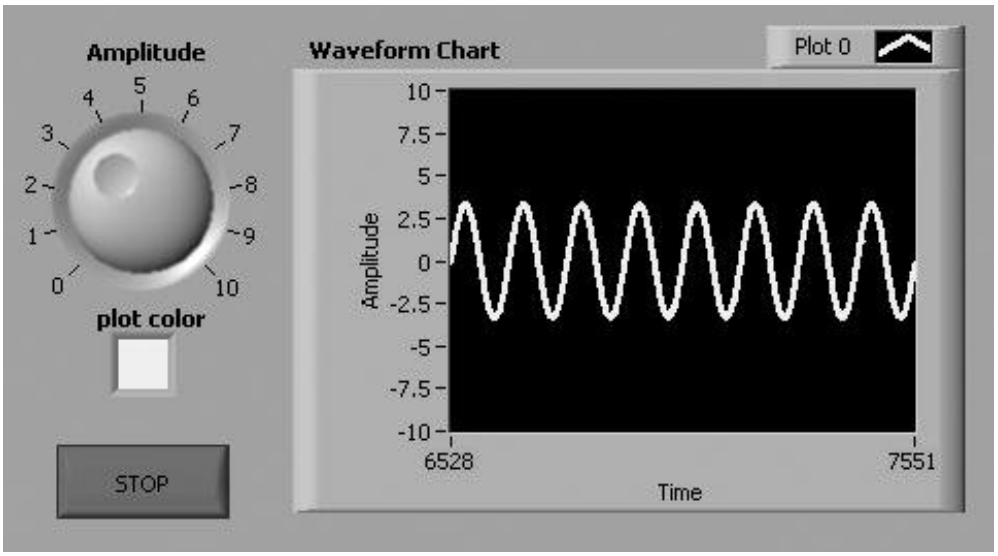
opposite behavior from a Case Structure, which will automatically configure output terminals with this setting turned off). You can change the value of an output terminal's Use Default if Unwired setting from the terminal's pop-up menu. We used the Use Default if Unwired setting to our advantage in the `stop` *button example, but this setting is not always desired. For example, when wiring a shift register through an Event Structure (as we did for the* `Work Done` *value), we do not want the shift register to be reset to the default valuewe want the shift register value to remain intact, until we explicitly change it.*

Activity 13-7: Using the Event Structure to Read Data Value Changes

In this activity, you'll use the Event Structure to control a simple loop that plots a sinewave. The Event Structure will detect if the stop button or the plot color changes its value from the front panel, and take appropriate action.

1. Create a VI with a waveform chart, a knob, a framed color box (the framed color box is a numeric control found on the Modern>>Numeric palette), and a stop button. Label the knob "Amplitude" and the color box "Plot Color," as shown in Figure 13.54.

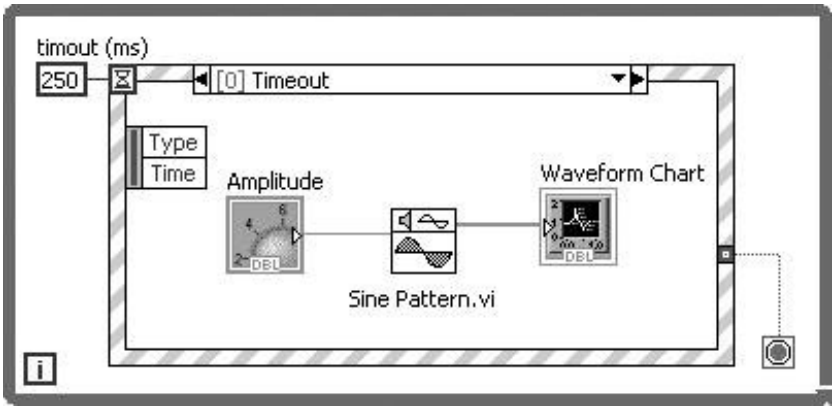
Figure 13.54. Front panel of the VI you will create during this activity



2. On the block diagram, create an Event Structure and put it in a While Loop.

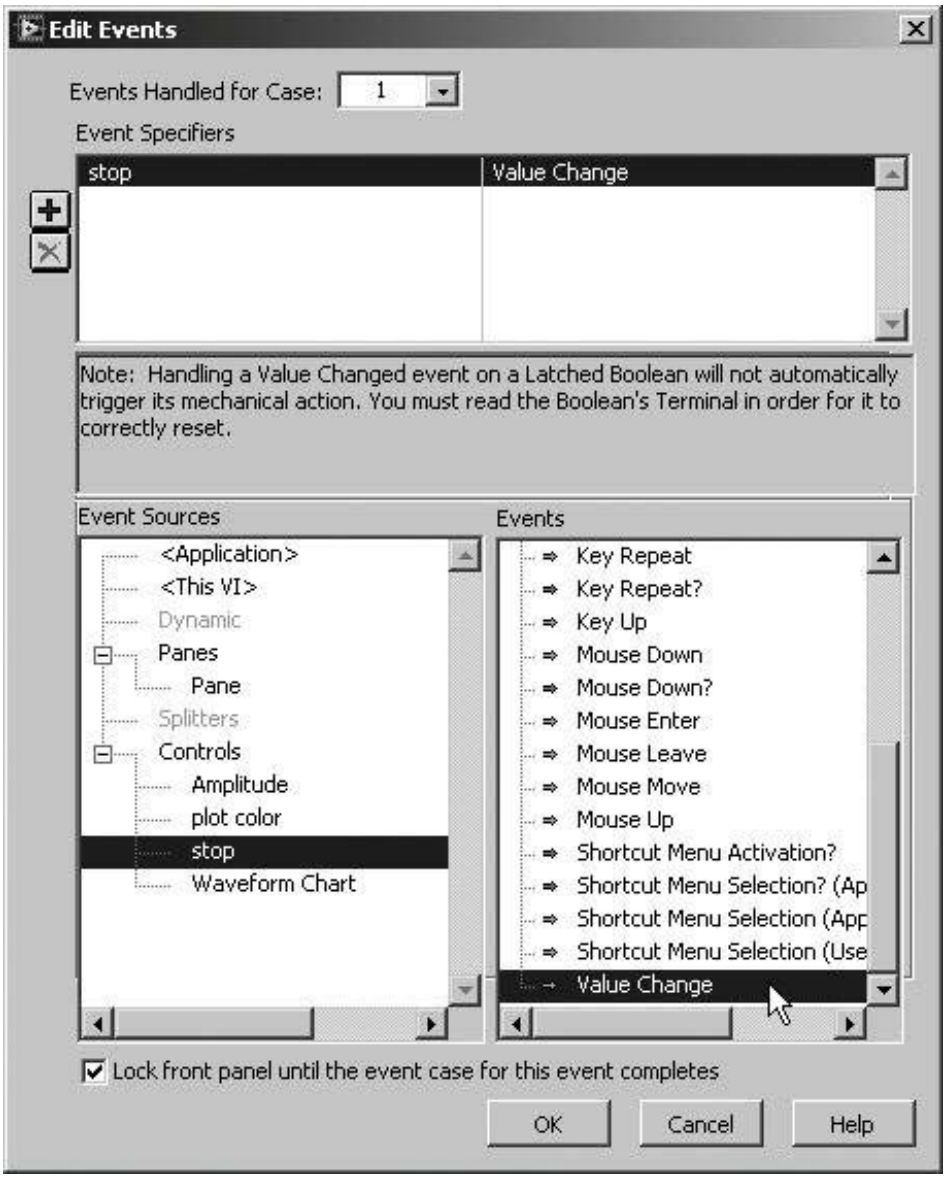
3. Your default Event will be the Timeout event (see Figure 13.55). Place a Sine Pattern function (from the Signal Processing>>Waveform Generation palette) here and connect the Amplitude knob to the "Amplitude" input. Connect the output to the chart. Set the Timeout value to 250 ms.

Figure 13.55. Block diagram of the VI you will create during this activity, showing the Event Structure's Timeout event



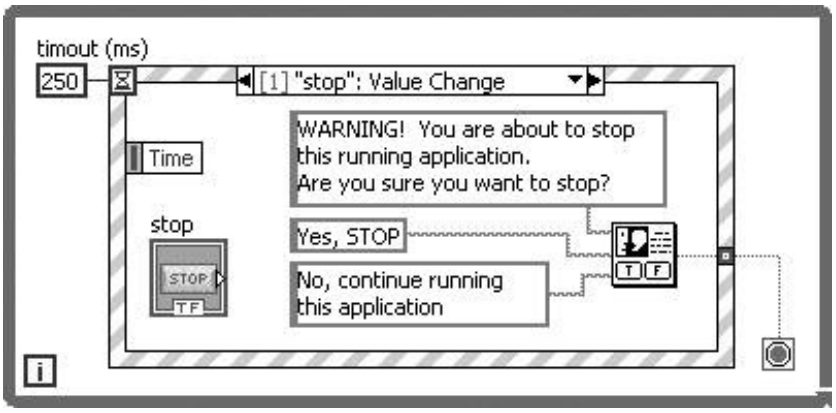
4. Now, add an event to detect if the "STOP" button is pressed. To add this event, pop up on the Event Structure and select Add Event Case . . . . This will open the Edit Events dialog, shown in Figure 13.56. Select the Controls>>stop from Event Sources and on the right column (Events), you'll need to scroll all the way down to select Value Change. Hit OK.

Figure 13.56. Adding an event



5. You've just created your "stop":Value Change event. Normally you'd just wire a Boolean to stop the VI here. But rather than exiting the VI directly if this happens, use a dialog to verify that the user really wants to stop. Give her the option of continuing on or really stopping. Can you do this without looking at Figure 13.57? Give it a try!

Figure 13.57. Block diagram of the VI you will create during this activity, showing the Event Structure's "stop":Value Change event



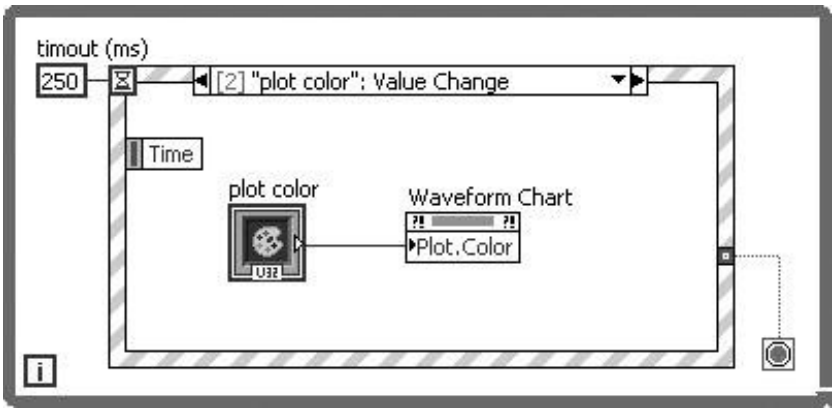
Two Button Dialog

Note how we used a Two Button Dialog function in this event. If the user presses the stop button, the dialog will execute. The Boolean output of Two Button Dialog tells us whether the user presses yes or no. Wire that Boolean output to the While Loop's Conditional Terminal, which should be configured as stop if true.

You must put the `stop` *button's terminal inside the event case, if you want the button to rebound to FALSE after the button press event is detected. The Latch When Release (as well as the Latch When Pressed) mechanical action will only reset the Boolean control when its terminal is read on the block diagram, and you must put it inside of the event case for it to be read after the button is pressed.*

6. Finally, add a third event that detects if the color box "plot color" changes value. Make it change the plot's color on the waveform chart to match. Try to do this by yourself without peeking at Figure 13.58 (hint: you'll need to use a property node for the chart).

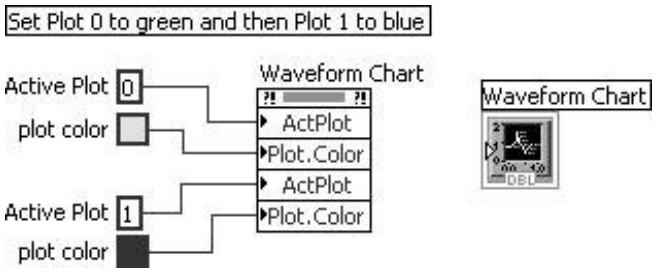
Figure 13.58. Block diagram of the VI you will create during this activity, showing the Event Structure's "plot color":Value Change event



The "plot color" terminal is wired to a property node we created for the waveform chart. The property we selected was Plot.Color, and we changed it to "write-mode" (by default, this property will come up as read-mode; you must pop up on it and select "Change to Write").

If the chart had multiple traces (plots), we would have had to set the Active Plot property before the Plot.Color by adding that property above Plot.Color in the property node, as shown in Figure 13.59. Note that a property node executes from top to bottom, setting each property in sequence. Keep this trick in your toolboxyou will definitely need it!

Figure 13.59. Setting the plot color of plot 0 and then the plot color of plot 1, by first setting the Active Plot property before setting the active plot's color



7. Save the VI as `Event Structure Data.vi`.

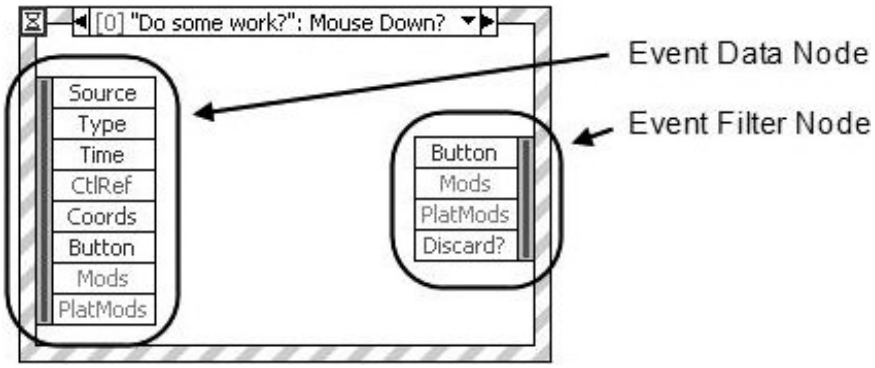
8. Run the VI and test your stop button and plot color box.

The next couple of sections briefly discuss some fairly advanced topics related to the Event Structure. Feel free to skip them if you feel you've had enough to digest in this rather meaty chapter. In simple applications, you often won't ever need to use these concepts.

Advanced Concepts: Event Data Node and Event Filter Nodes

If you were paying close attention, you probably noticed that there are terminals inside the event cases, attached to the left and (if you ever chose an event with a red arrow next to its name in the Edit Events dialog) right walls, that look very similar to the Unbundle By Name and Bundle By Name functions (but without any cluster wires connected to them). These collections of terminals are called the Event Data Node and the Event Filter Node (see Figure 13.60), and are used for accessing and modifying data related to the event.

Figure 13.60. Event Structure with its Event Data Node and Event Filter Node showing



The Event Data Node provides you with extra information about the event, such as when it happened, what type of event it was, mouse coordinates (for mouse-related events), and more.

We'll talk briefly about the Event Filter Node now.

Advanced Concepts: Notify Events Versus Filter Events

Up to this point, you may not have ever seen the Event Filter Node, shown in Figure 13.60. So far we have only seen examples that do not use this node. In order to explain when the Event Filter Node will appear, we first need to understand that there are two different types of events: *notify events* and *filter events*. Only filter events have an Event Filter Node.

We can distinguish notify events from filter events in the Edit Events dialog from the color of the arrow icon to the left of the event name in the Events listboxnotify events have a *green* arrow next to them, and filter events have a *red* arrow next to them. You might also notice that events whose name ends in a question mark are almost always filter events; however, there are many filter events whose name does not end in a question mark.

They are called filter events, because you can choose to filter the event and/or modify the event data by wiring data to the Event Filter Node terminals. What this means is you can "catch" the event and, if you wish, modify the event or even tell your VI to ignore the event.

For example, in Figure 13.60, we could discard the Mouse Down event by wiring a value of True into the Discard? Input of the Event Filter Nodethe result would be that the user's mouse down would not cause the button to change state, thus making the button behave as if it were disabled.

In order to filter an event, the event case must run synchronously with the user interface. This means that the user interface will be *locked* from the time the event occurs until the event case that handles it finished executing. (That is why, in the Edit Events dialog, the `Lock front panel until the event case for this event completes` checkbox is always checked for filter events.) This contrasts sharply with notify events, which only receive notification that the event occurred. They cannot change the event data, or discard the event, so there is usually no reason to lock the user interface while a notify event case is executing (although you can select the `Lock front panel until the event case for this event completes` setting, if you like). Because notify event cases never change the event data (or filter/discard the event), they can run asynchronously from the user interface.

Advanced Concepts: Dynamic Events and User Events

So far, we have seen how to use the Edit Events dialog to configure an event case to handle events for controls on the front panel of the VI that contains the Event Structure. However, the Event Structure also has a feature called *Dynamic Events*, which allows you to programmatically specify which controls will be handled by the event cases. You can pop up on the frame of the Event Structure and select Show Dynamic Event Terminals to show the terminals to which you can wire an Event Registration Refnum created using the Register For Events node (found on the Programming>>Dialog & User Interface>>Events palette, shown in Figure 13.61).

Figure 13.61. Events palette

The Register For Events node can be used to register both VI Server object events and User Events. We will learn about VI server in Chapter 15. A User Event (containing event data that you define, similar to a queue or notifier) is created and generated using the Create User Event and Generate User Event functions, respectively (found on the Programming>>Dialog & User Interface>>Events palette).

An in-depth look at Dynamic Events and User Events is beyond the scope of this book. But, if you are interested in learning more about these advanced event features, take a look at the help resources available for functions and structures on the Programming>>Dialog & User Interface>>Events palette, as well as the examples that ship with LabVIEW in `examples\general\dynamicevents.llb`.

PREV NEXT

Type Definitions

Type definitions in LabVIEW are a powerful tool for defining your control and indicator data types in your program. A type definition, often called *typedef* for short, is a LabVIEW custom control file (`.ctl`), that is configured to act a "master template" for the data type of *slave* controls that have been linked to the typedef. We'll learn about customizing the appearance of controls in Chapter 17, "The Art of LabVIEW Programming," later on, but right here we'll focus on the data type definition functionality of custom controls.

Typedefs are among the most useful structures in the LabVIEW programmer's arsenal, and perhaps the best way to introduce them is to start with an example. Suppose you have a program that you built for controlling a motor. In your LabVIEW program, you've created a cluster called `Motor State` that contains information about the motor's power state (on or off), its speed, and the direction it is moving (such as the one shown in Figure 13.62).

Figure 13.62. Motor State cluster

Now let's say your application uses this cluster in several subVIs. Perhaps you have a subVI to check the motor status, another one to turn the motor on or off, and another one to switch directions. On the front panel of each subVI (shown in Figure 13.63), you're using a cluster like the one shown in Figure 13.62.

Figure 13.63. Three subVIs that use the Motor State cluster

[View full size image]

Now let's say that after some time of your program running fine, you add a temperature sensor to the motor. You now want to monitor temperature in various places in your program, so you will need to add temperature to your `Motor State` cluster. Your `Motor State` cluster now needs to look like the one shown in Figure 13.64.

Figure 13.64. Motor State cluster, after adding a **`Temp`** numeric indicator

If you didn't know about typedefs when you wrote your original program, you would need to go and edit every single cluster that contains the motor state and "temperature." Not only is this tedious, but it can be an error-prone activity that introduces bugs into your application (and that's not good). And in some applications, you might have dozens or even hundreds of places the cluster needs to change (it will take a lot of time to do all that work). And then you might need to change the cluster contents again, later (ugh)!

The solution is to first create a typedef for the `Motor State` cluster and use *instances* of this typedef in your various VIs. Because the instance are linked to the typedef and slaves to its data type, if you ever need to change that `Motor State` cluster, you only have to modify the typedef, and all the VIs that use the typedef will automatically be updated to the new data type for you!

We'll guide you through the process of creating a typdef, step-by-step, in the following activity.

Activity 13-8: Creating a Typedef

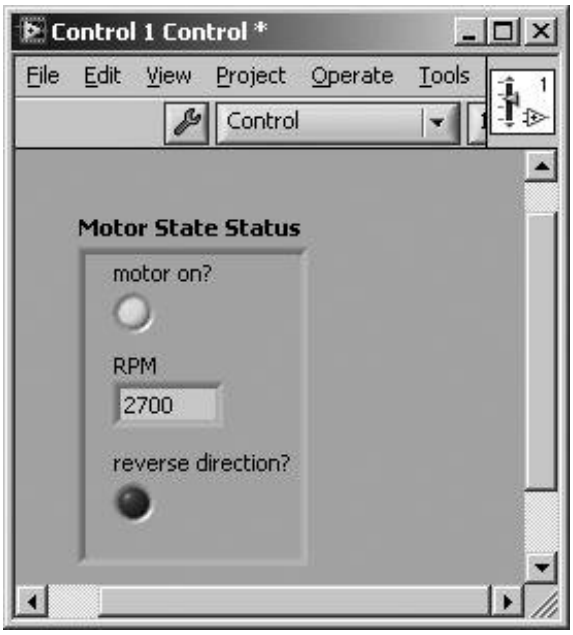
In this activity you will create a type definition that can be used in several locations of your application. Each location that uses it will be updated automatically when the type definition is edited.

1. In a new blank VI, create a cluster like the one shown earlier in Figure 13.64.

2. Now pop up on the cluster (be sure you pop up on the cluster and not a cluster element by right-clicking on the cluster frame), and select Advanced>>Customize . . . . Alternately, you can select the cluster with the positioning tool, and from the Edit menu, select Customize Control . . . .

3. You will now see your cluster in a new window called Control 1 Control or something similar (see Figure 13.65). This is a custom control window.

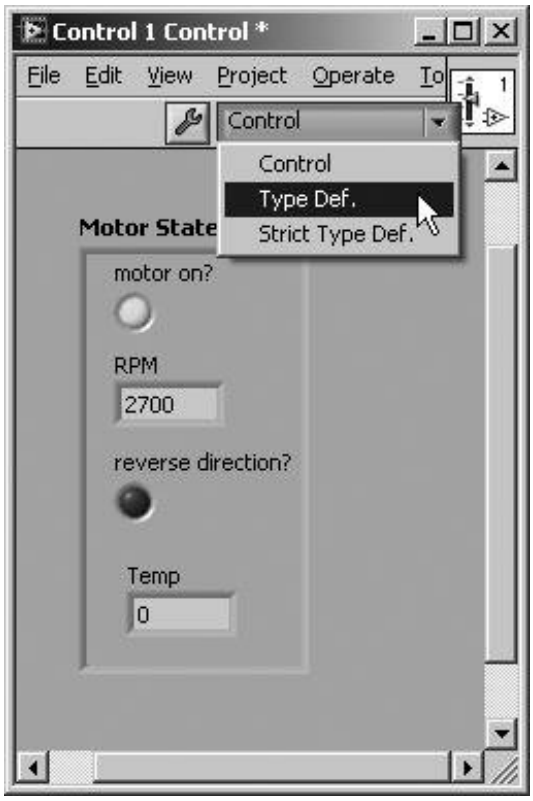
Figure 13.65. Control editor window showing the custom control type definition you will create during this activity



4. The control window is like a front panel with no block diagram for a single control. You can edit and customize your control, but there is no functionality to program here. Go ahead and add an element to the "Motor State" cluster called "Temp."

5. The Control window has a drop-down menu called Type Def status. By default it is set to "Control." Click on this and choose "Type Def," as shown in Figure 13.66.

Figure 13.66. Configuring the custom control as a Type Definition

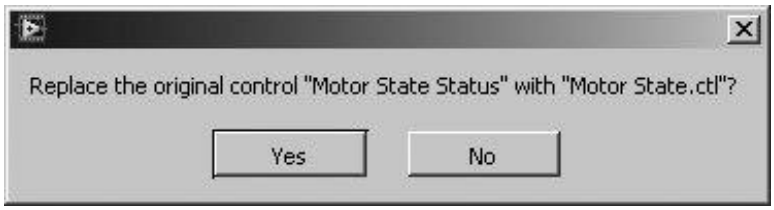


When creating Type Definition, be sure and remember to set the option to "Type Def." If you leave it set to "Control" in the Control Editor window, it will not update the instances of the control in your VIs!

6. Now save your type definition. Because a type definition is really a custom control, LabVIEW saves these as control files, with a `.ctl` extension. Save this as `Motor State.ctl`.

7. You've just created a typedef. Now close this window. When you do so, you will get a dialog like the one in Figure 13.67, asking you if you want to replace your original control with Motor State.ctl. Say yes.

Figure 13.67. Confirmation dialog asking if you would like to replace the original front panel control with your new type definition



8. Now look at the original VI in which you had the `Motor State` cluster. It should have automatically changed to include the new "Temp" element you added when you were editing the control.

9. To see how you can now use your type definition control, open a new VI. To insert the typedef, click on "Select a Control . . ." from the Controls palette, as shown in Figure 13.68.

Figure 13.68. Choosing Select a Control . . . from the Controls palette to browse to a custom control file that you would like to add to the front panel

10. The ensuing dialog box will let you browse for LabVIEW control files (`.ctl` files). Find the "Motor State.ctl" typedef you made a minute ago and select it. You will now see the typedef control on your front panel.

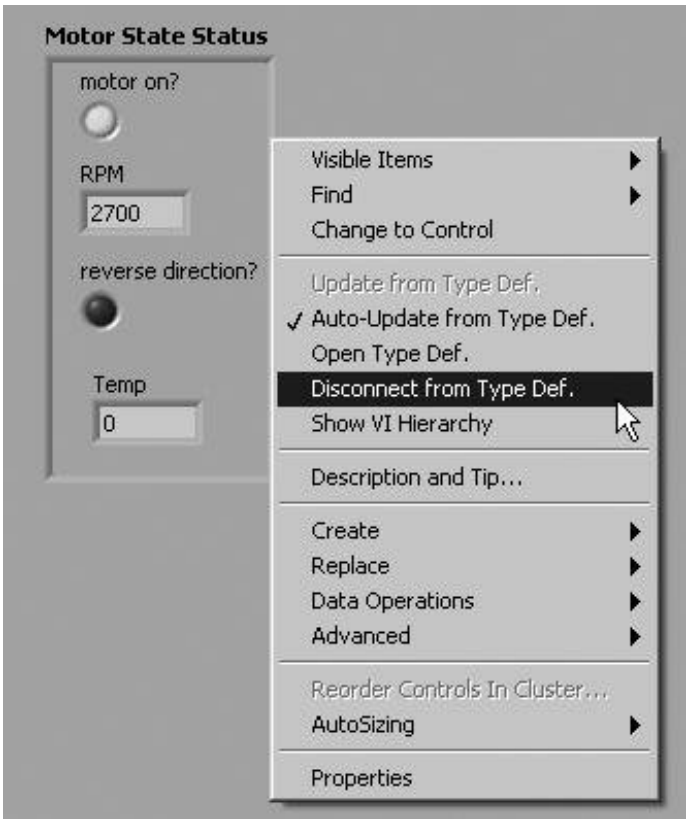
11. Now if you modify the Motor State.ctl again, all the VIs that use this typedef will be updated.

12. If you want to play with this typedef and VIs that have it on its front panel, we've provided it for you on the CD in the `EVERYONE\CH12` directory. Look for the "Type Def Example" directory there.

Remember that a type definition is like a "master template." You can have any number of instances of controls linked to that typedef, so that if you modify the typedef, the controls get modified as well, automatically by default.

If you have a control on a front panel that is tied to a typedef, you can choose to pop up on the control to see some options related to the typedef (see Figure 13.69).

Figure 13.69. Disconnecting a control from its type definition "master"



These options follow:

- Update from Type Def. This is disabled whenever "Auto-Update from Type Def." is checked, which is by default. If the auto-update option is not checked, select this to force the control to update itself from the type definition.
- Auto-Update from Type Def. When this is checked, the control automatically updates itself any time the typedef changes.

- Open Type Def. This will cause LabVIEW to open the typedef `.ctl` file so you can see it or edit it.
- Disconnect from Type Def. This disconnects the control from the type definition. Once you do this, any changes to the type definition will have no effect on this control. You cannot "reconnect" it, either. To go back, either choose Undo (<ctrl-Z> [Windows], <command-Z> [Mac OS X], or <meta-Z> [Linux]) or delete the control and re-insert an instance of the typedef.

Type definitions are just one kind (but arguably the most important kind) of LabVIEW custom controls. The other kinds are "controls" proper and "strict type definitions." A strict type definition is like a type definition except that it also forces the appearance, and not just the data type, of its linked controls to stay the same. We'll talk more about custom controls in Chapter 17. There is also more to typedefs than we've been able to cover here, but for further reading, consult the LabVIEW documentation or some of the resources listed in Appendix E, "Resources for LabVIEW."

In general, you should develop the habit of creating typedefs in your applications whenever you think you might be re-using a control or a data structure such as clusters. It will save you hours of work and make your code easier to maintain.

Type definition instances are often placed on the front panel as controls and indicators. However, you can also place them on the block diagram as constants. You will see an example of this when we discuss the Standard State Machine design pattern later in this chapter. To open the type definition of a constant, do just as you would for a control or indicator: pop up on the constant and select Open Type Def.

PREV NEXT

The State Machine and Queued Message Handler

In Chapter 6, "Controlling Program Execution with Structures," you learned about the power of the While Loop + Case Structure combination. We saw a few permutations of this simple application design pattern. We ended the discussion by providing a scalable solution for handling multiple button pushes.

But there is an even more powerful pattern for combining the While Loop and Case Structure: one that allows one frame of a case structure (inside of a While Loop, of course) to define one or more cases that will be executed on the subsequent iterations of the While Loop. There are a variety of permutations of this pattern, but they are commonly referred to as "state machines" and "queued message handlers."

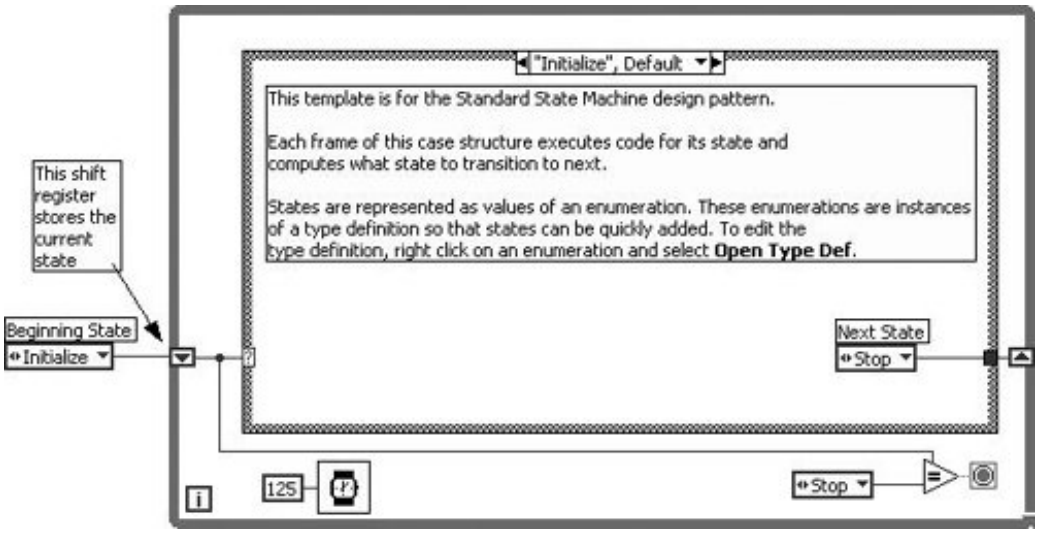
There are a huge variety of implementations of state machine patterns. Fortunately, LabVIEW comes with some very good example templates that you can use in your applications. Just launch the File>>New . . . dialog to find thesedo that now, and we'll guide you through two very useful example templates.

The Standard State Machine

If you haven't already, open the File>>New . . . dialog from the menu. Browse to VI>>From Template>>Frameworks>>Design Patterns, select Standard State Machine, and press OK. This will open an untitled VI with a block diagram that looks like the one shown in Figure 13.70. (Actually, yours will look larger, as we have resized some of the block diagram objects so that they would fit compactly on this page.)

Figure 13.70. Block diagram of the Standard State Machine template

[View full size image]



There are some key concepts related to this state machine template:

1. A shift register is used to store the *state variable*, which is simply a value that determines which frame of the state machine will execute.

2. The state variable data type is an enum that has a type definition. (Type definitions were discussed in the last section of this chapter.) Every enum constant you see on the block diagram is an instance of the type definition. You cannot edit the enum constants. You must edit the type definition in the Control Editor window, which may be opened by selecting Open Type Def. from the pop-up menu of any of the enum constants. This ensures that all the enum constants' elements are identical, so that you don't have to edit each of them manually when making changes. (Note that when you attempt to save the new untitled state machine VI, LabVIEW will ask that you first save the enum type definition (`.ctl`) file.)

3. Because the state variable is an enum and is wired to the case selector terminal of the Case Structure, the case names are strings that are validated by the LabVIEW editor. There is no chance of misspelling a case nameif you do so, the VI's run button will appear broken and you will not be able to run the VI.

4. The conditional terminal of the While Loop is wired to a comparison function that evaluates whether the state variable is equal to "Stop." If so, the While Loop terminates, and the state machine stops.

5. Each frame of the case structure must pass out a value for the state variable, which is wired to the shift register, called the *state variable shift register*. This allows each case to define which case will execute on the next iteration. (Except, the "Stop" case does not get to decide which case executes on the next iteration, because the While Loop will terminate on the same iteration as the "Stop" case. Because each case (or *state*) can only define the state on the next iteration (it cannot, for example, "queue up" more than one state to execute), the LabVIEW Standard State Machine template is technically called a *Finite State Machine*.

6. The state variable shift register must be initialized outside of the loop to define the initial state of the state machine. Otherwise, the state machine will have a memory and *always* execute the

"Stop" case after the first time it is calledthat would be bad.

We recommend having no Default case in your Case Structure. Because the state variable data type is an enum, the Case Structure does not need a Default framethere are a small, finite number of possible state variable values. Having a Default frame just means that you might forget to add a new case to the Case Structure after you add a new element to the state variable enum. If you don't have a Default frame, your VI will be broken if there is not a case to handle each element of the enum. If you leave the "Initialize" case as the Default case, then the "Initialize" frame will execute if the state variable value is not handled by a casethat's not the desired behavior.

Now you'll do an activity that uses the State Machine template to do something useful.

Activity 13-9: Using the Standard State Machine

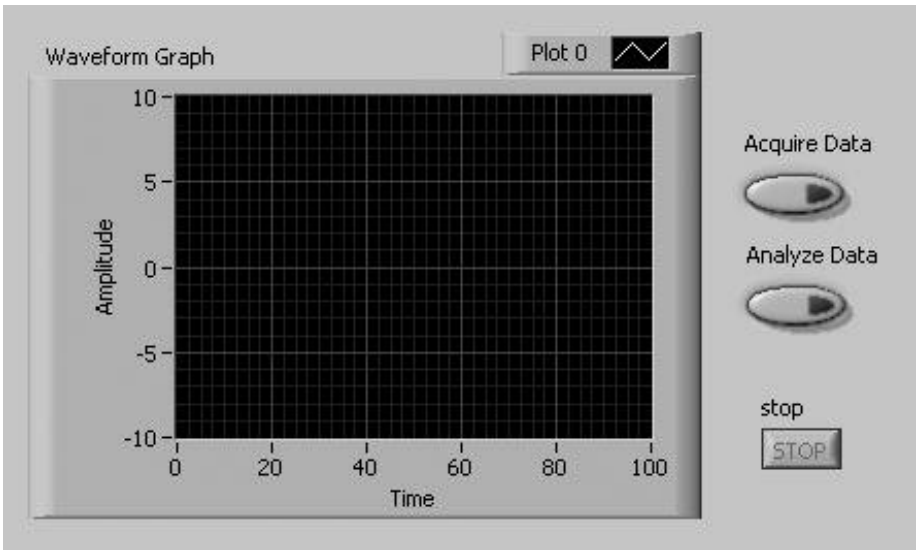
In this activity you will create a state machine from the Standard State Machine template that ships with LabVIEW.

1. Open a new VI, created from the Standard State Machine template. Do this by selecting File>>New . . . from the menu, browse to VI>>From Template>>Frameworks>>Design Patterns in the resulting dialog, selecting Standard State Machine, and pressing OK.

2. Save the VI as `State Machine.vi` and, when prompted, save the StateMachineStates enum type definition as `State Machine States Enum.ctl`.

3. Your front panel should look like the one shown in Figure 13.71. Ensure that each of the Buttons are configured for Mechanical Action>>Latch When Released so that they rebound to FALSE after their terminals are read on the block diagram.

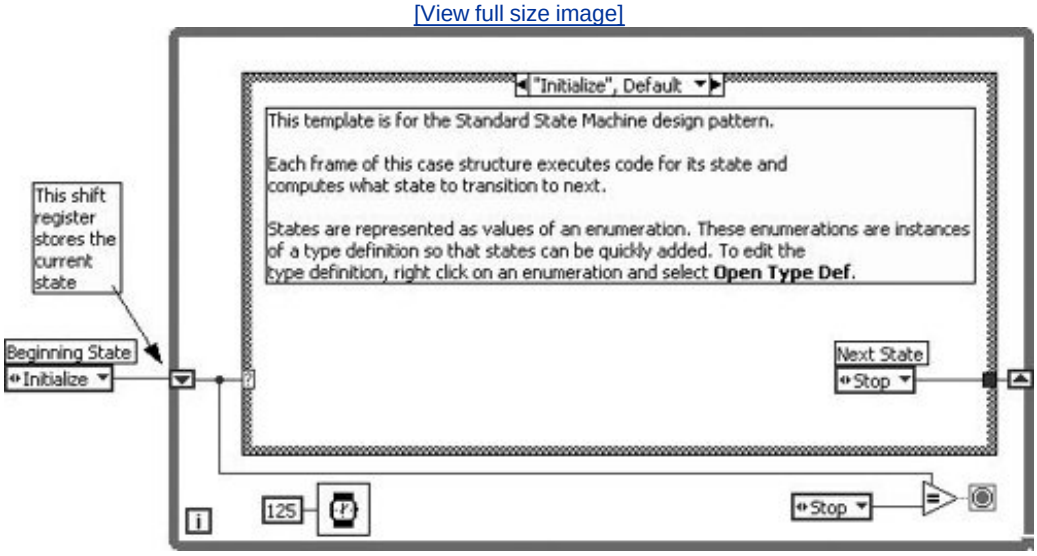
Figure 13.71. Front panel of the VI you will create during this activity



4. Your block diagram should initially look like the one in Figure 13.72.

Figure 13.72. Block diagram of the VI you will create during this activity, in its initial state

[View full size image]



5. Pop up on the Case Structure and select Remove Default so that the Case Structure has no Default (catch all) frame. With an enum-type Case Structure, a Default frame is unnecessary and a liability. We would rather have a broken VI, than execute the default frame by accident, if we accidentally did not have a case for every possible enum value.

6. Edit the state variable enum type definition, so that it has the following items:

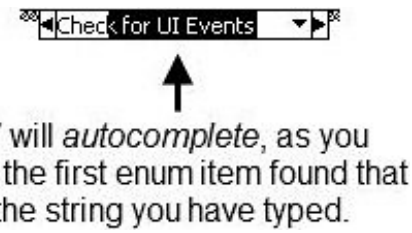
   - Initialize
   - Stop
   - Check for UI Events
   - Acquire Data
   - Analyze Data

   Ensure that the enum item names are in the same order shown above.

7. In the "Initialize" case, change the state variable enum constant's value from "Stop" to "Check for UI Events." (This constant is a type definition that you will need to edit, as you just learned in the "Type Definitions" section of this chapter.) This will cause the state machine to execute the "Check for UI Events" case immediately after initialization. (Note we will be coming back to the "Initialize" case later to add some code that initializes our application.)

8. Make the "Stop" case visible and then pop up on the Case Structure and select Add Case After. The new case should be automatically named "Check for UI Events." If it is not, use the Labeling tool to rename it. Note that LabVIEW auto-completes the case name, as shown in Figure 13.73.

Figure 13.73. LabVIEW's auto-completion of case names as you type them into the Case Structure



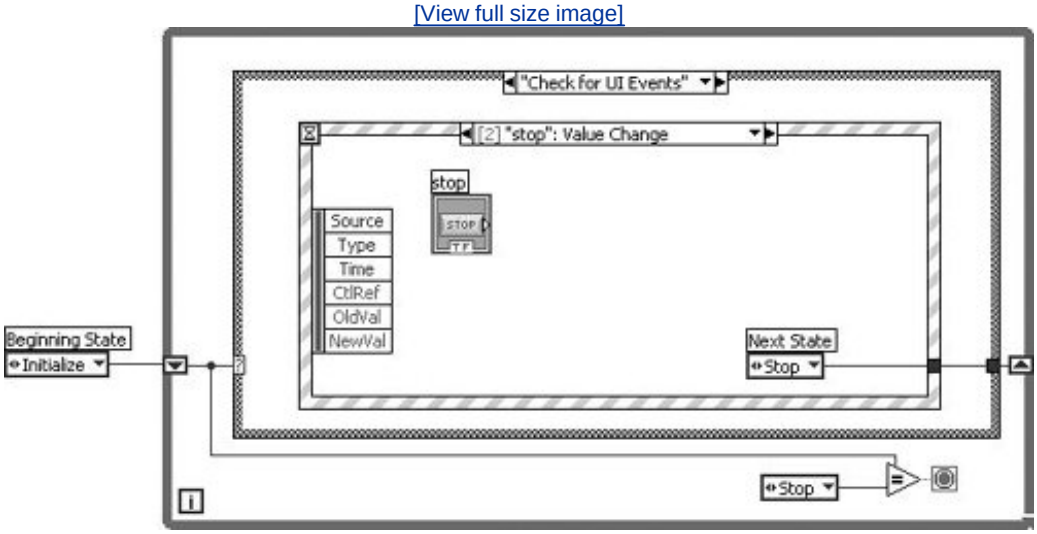
9. In the "Check for UI Events" case, place an Event Structure, as shown in Figure 13.74. Remove the "Timeout" event case. The Event Structure should have the following event cases:

   - "Acquire Data": Value Change
   - "Analyze Data": Value Change
   - "Stop": Value Change

   In each of these event cases, place the respective (like named) Buttons.

Figure 13.74. Your VI's block diagram after adding the "Check for UI Events" case

[View full size image]



Again, ensure that each of the Buttons are configured for Mechanical Action>>Latch When Released so that they rebound to FALSE after their terminals are read on the block diagram.

10. In each of the event cases, place an instance of the state variable enum as a constant. Its value should correspond to the name of the button press. For example, the "Stop": Value Change event case should output an enum value of "Stop," the "Analyze Data": Value Change event case should output an enum value of "Analyze Data," and so on.

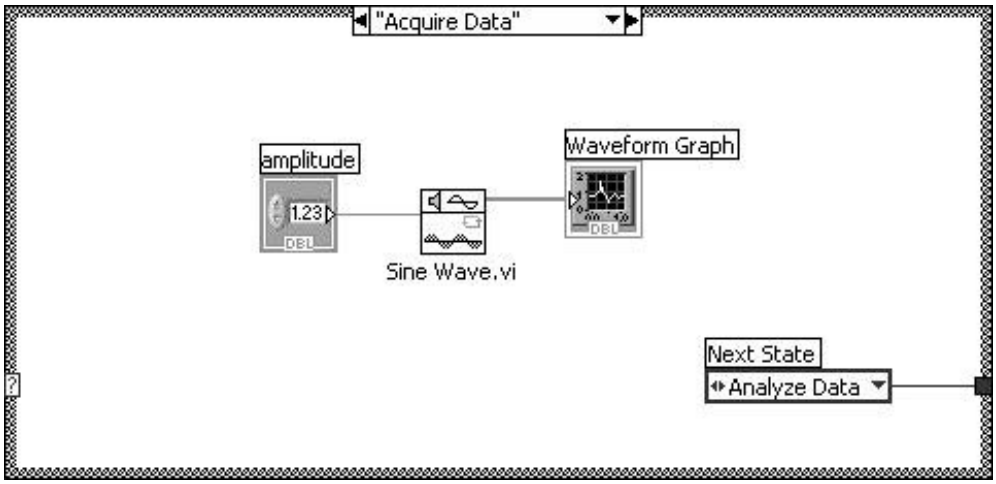
The block diagram should now look similar to the one shown in Figure 13.74. Note that the time has been removed. It is not necessary because the event structure will wait patiently for any front panel events to occur.

11. Pop up on the output tunnel (where the state variable enum is passed out) and uncheck the "Use Default If Unwired" option. This is a huge liability, if we should forget to wire the state variable out of one of the event cases. We do not want it to reinitialize the state machine (which it would, because "Initialize" is the 0^{th} [and therefore default] item of the enum). With this option turned OFF, we will get a broken VI, telling us that we forgot to wire the state variable enum out of an event case, should we forget.

12. Add a case to the Case Structure for "Acquire Data," as shown in Figure 13.75. Note that the state variable that is output from this case is "Analyze Data." This is a perfect example of the power of the state machine. The Acquire Data case will execute next, before it subsequently goes back to the "Check for UI Events" case. This is the power of the state machine pattern.

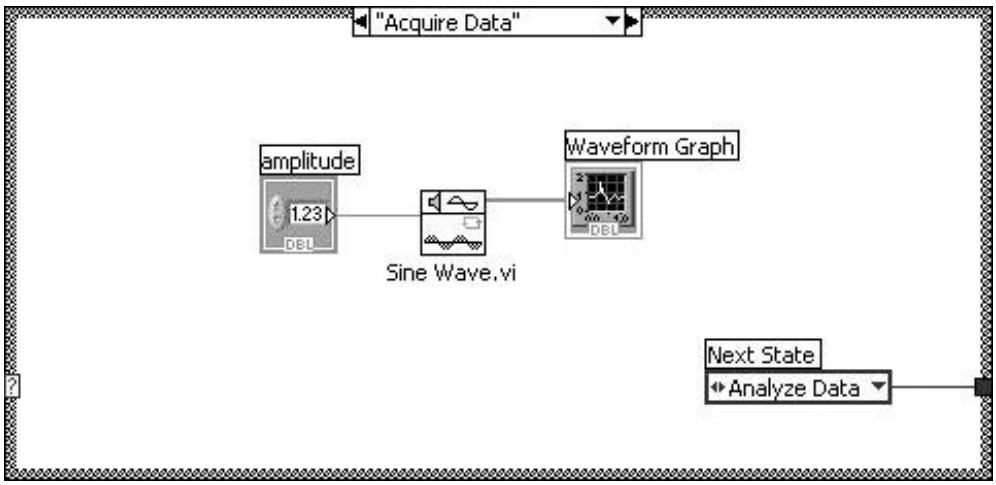
Figure 13.75. Your VI's block diagram after adding the "Acquire

Data" case



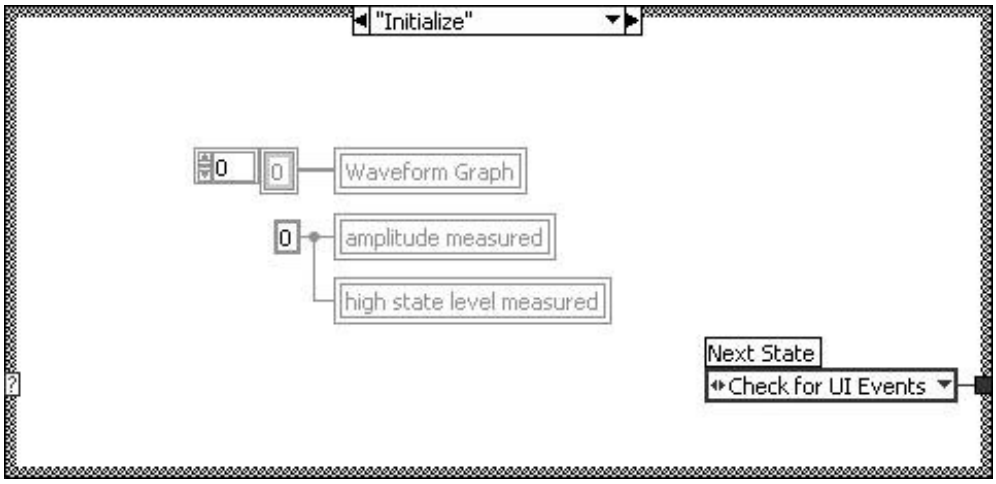
13. Add a case to the Case Structure for "Analyze Data," as shown in Figure 13.76. You will need to read the data from the waveform graph using a local variable in read mode.

Figure 13.76. Your VI's block diagram after adding the "Analyze Data" case



14. Modify the "Initialize" case, adding code that writes initial data to the local variables of the waveform graph and signal measurements, as shown in Figure 13.77. Note that the array constant is an empty array. You can ensure that it is empty, by selecting Data Operations>>Empty Array from its pop-up menu.

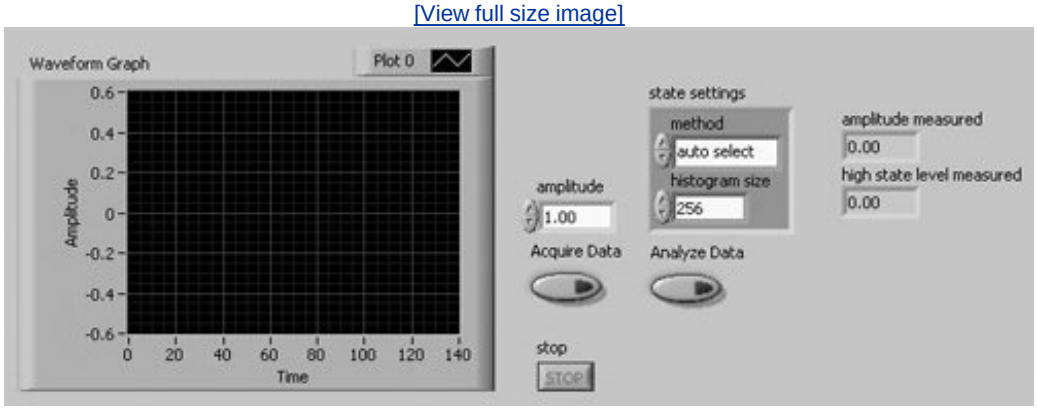
Figure 13.77. Your VI's block diagram after adding the "Initialize" case



Your front panel should now look similar to Figure 13.78.

Figure 13.78. Your VI's front panel after completing this activity

[View full size image]



15. Save your VI as `State Machine.vi`. You are now ready to run it.

Run your VI. Press the acquire button and note that the data are displayed in the waveform graph and the analysis parameters are also updated (remember, each time you acquire data, the analysis will *also* be performed).

Change the analysis `method` from "peak" to "histogram," and then press the `Analyze Data` button. Notice how the measured values change.

Change the `amplitude` value, and then press the Acquire button again. Note that the amplitude of the signal on the graph changes, as do the analysis results.

Press the `Stop` button, and then run your VI again. Note that the graph and analysis results are initialized to the values you specified in the "Initialize" case.

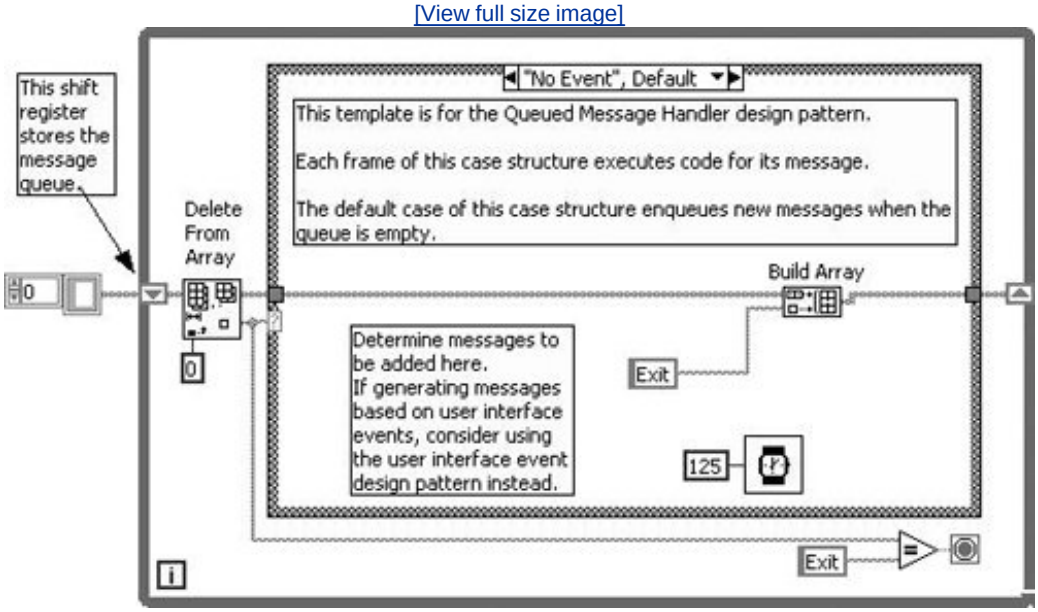
Congratulationsyou've just made a very powerful state machine.

The Queued Message Handler

Open the File>>New . . . dialog from the menu. Browse to VI>>From Template>>Frameworks>>Design Patterns, select Queued Message Handler, and press OK. This will open an untitled VI with a block diagram that looks like the one shown in Figure 13.79. (Actually, yours will look larger, as we have resized some of the block diagram objects so that they would fit compactly on this page.)

Figure 13.79. Block diagram of the Queued Message Handler template

[View full size image]



There are some key concepts related to this Queued Message Handler template:

1. A shift register is used to store the *message queue*, which is an array of message strings.
2. Each iteration of the While Loop, the Delete From Array function "pops" the last element off of the array (from the end of the array, because the index terminal of Delete From Array is unwired). This makes it a *Last In First Out (LIFO)* queue.

We do not like the LIFO queue behavior. We prefer to have the queue elements removed from the front of the queue, making it a First In First Out (FIFO) queue. Making this change is easyjust wire a zero (0) constant to the index terminal of the index input terminal of Delete From Array.

3. The message queue data type is an array of message strings. Each message string corresponds to a case of the Case Structure. The messages in the message queue are executed by the queued message handler in sequence (starting from the last element of the array and working up to the 0^{th} element).

4. Because the *message* element type is a string and is wired to the case selector terminal of the Case Structure, the case names are strings that are *not* validated by the LabVIEW editor. There is a very good chance of misspelling a case nameif you do so, the VI will still run.

We suggest that you make the Default frame an error-generating frame that displays a message (or generates an error) when an unhandled message is passed into the Case Structure. This will alert you to the bug in your code, and help you fix it.

5. The conditional terminal of the While Loop is wired to a comparison function that evaluates whether the message string is equal to "Exit." If so, the While Loop terminates, and the queued message handler stops.

6. Each frame of the case structure must pass out the message queue, which is wired to the shift register, called the *message queue shift register*. This allows each case to modify the message queue, either adding or removing message elements to the array. These elements may be added to the front or back of the queue, using the Build Array function.

7. The message queue shift register must be initialized outside of the loop to define the initial messages on the queue of the queued message handler. Otherwise the queued message handler will have a memory and *always* execute the "Exit" case after the first time it is calledthat would be bad. Generally, the message queue is initialized with a single array element whose value is "Initialize."

As you can imagine the queued message handler (QMH) is orders of magnitude more powerful than the standard state machine (SSM), because every case has full control of the entire message queue.

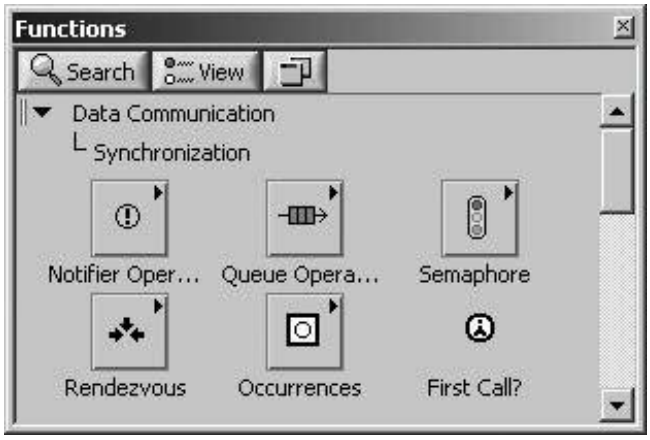
Remember that the SSM cases can only define the *next* case that will execute. The QMH is naturally more finicky, due to the loose typing of the message element's string data type. This can be improved by converting the message element into a type definition enum, similar to the one used by the SSM. Finally, it should be mentioned that with the extreme flexibility of allowing each case to have full control of the message queue can make debugging a challenge. It can also reduce the determinism of your code. We believe that artificial intelligence could be implemented using a QMH . . . they can sometimes have a mind of their own. :-)

PREV NEXT

Messaging and Synchronization

In the last section, you learned about the queued message handler, which allows you to enqueue string message elements onto an array of string messages. This is a very powerful tool for application programming. But, LabVIEW also has some built-in tools for messaging and synchronization that allow you to implement some even more powerful applications. The Data Communication>>Synchronization palette (shown in Figure 13.80) contains the building blocks for complex, parallel, application logic, where dataflow becomes a little more fuzzy. These are the Queue, Notifier, Semaphore, Rendezvous, Occurrence, and First Call? VIs and functions.

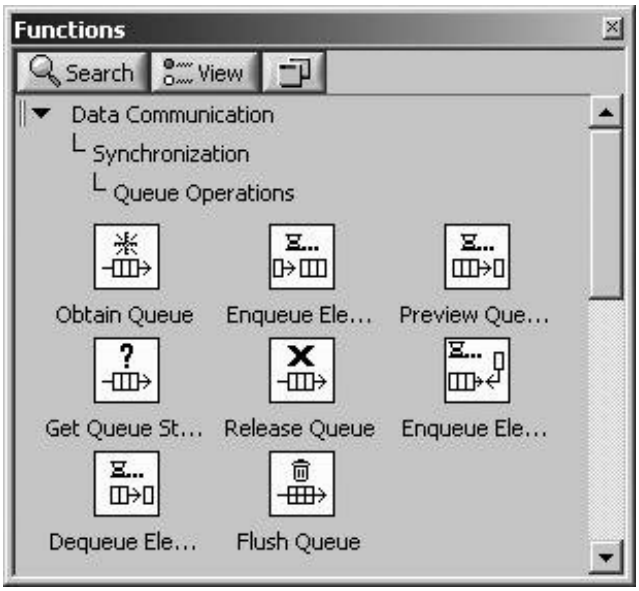
Figure 13.80. Synchronization palette



Queues

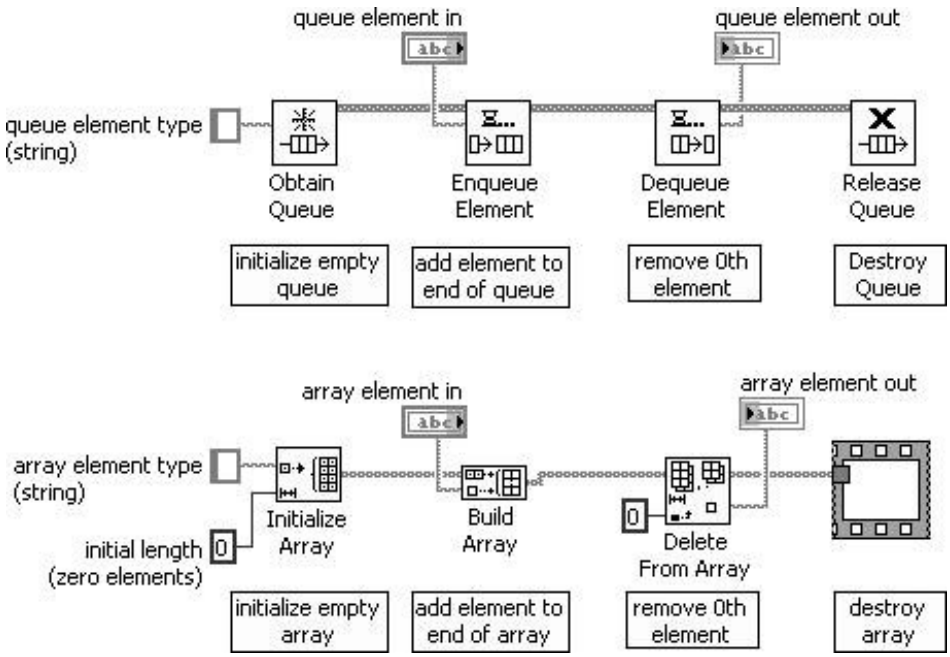
Queues are one of the most commonly used messaging constructs. As we learned in the last section, where we discussed the queued message handler, a queue is an ordered set of elementsnamely, an array. To *enqueue* is to *add an element to the queue*, and to *dequeue* is to *remove an element from the queue*. The Queue Operations palette (shown in Figure 13.81) contains all the functions for operating on queues.

Figure 13.81. Queue Operations palette



The queue can be thought of as an array, which is operated on by referencemeaning, when you pass a queue around the block diagram of your VI using a wire, it is not the data of the queue that flows through the wire, but a reference to it. You cannot access the queue's array directly; you can only access the queue data through the various queue functions. To make the analogy easier to understand, see Figure 13.82 for a side-by-side comparison showing the analogous queue and array operations. You will learn more about the queue functions next. After learning about each of the queue functions, come back to this figure and study it to solidify this analogy.

Figure 13.82. Queues are simply arrays operated on by reference.

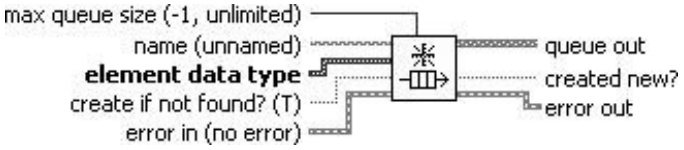


When you fork a queue reference wire, the queue is not copied. Rather, the reference is copied and both copies of the reference refer to the same queue. Many locations in your code can have a reference to the same queue; this is how they are able to share the queue and use it to communicate with each other.

Creating and Destroying Queues

In order to use queues, you will need to first create a queue. The Obtain Queue function (shown in Figure 13.83) creates a queue, or returns a reference to an existing (named) queue.

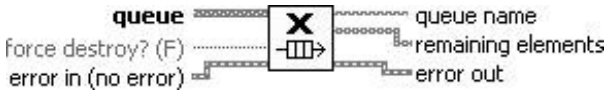
Figure 13.83. Obtain Queue



Obtain Queue (Data Communication>>Synchronization>>Queue Operations palette) returns a reference to a queue. You must specify the element data type, which is a polymorphic input that accepts any LabVIEW data type. This will define the data type used by the other queue functions when operating on the resulting queue. Use named queues to pass data between two sections of a block diagram or between two VIs. If you do not wire name, the function creates a new, unnamed queue reference. If you wire name, the function searches for an existing queue with the same name and returns a new reference to the existing queue. If a queue with the same name does not already exist and create if not found? is TRUE, the function creates a new, named queue reference.

Release Queue (Data Communication>>Synchronization>>Queue Operations palette) releases a reference to a queue (see Figure 13.84). If you are releasing a reference to a named queue (one that was given a name when created using the Obtain Queue function), in order to destroy the *named queue*, call the Release Queue function a number of times equal to the number of times you obtained a reference to the queue or stop all VIs using the queue reference.

Figure 13.84. Release Queue

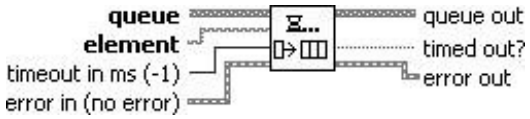


If force destroy? is TRUE, this function releases all references to the queue and destroys the queue.

Enqueuing and Dequeuing Elements

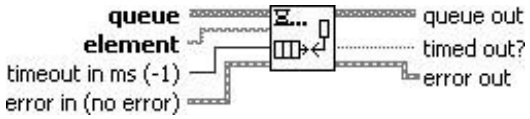
Enqueue Element (Data Communication>> Synchronization>>Queue Operations palette) adds an element to the back of a queue (see Figure 13.85). If the queue is full, the function waits timeout in ms before timing out. If space becomes available in the queue during the wait, the function inserts the element and timed out? is FALSE. If queue becomes invalid (for example, the queue reference is released), the function stops waiting and returns error code 1122. Use the Obtain Queue function to set the maximum size of the queue.

Figure 13.85. Enqueue Element



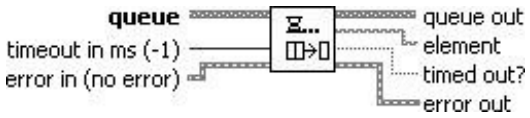
Enqueue Element at Opposite End (Data Communication>>Synchronization>> Queue Operations palette) adds an element to the front of a queue (see Figure 13.86). This function is similar to the Enqueue Element function. If the queue is full, the function waits timeout in ms before continuing. If queue becomes invalid (for example, the queue reference is released), the function stops waiting and returns error code 1122.

Figure 13.86. Enqueue Element at Opposite End



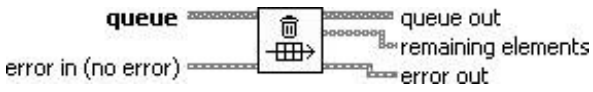
Dequeue Element (Data Communication>> Synchronization>>Queue Operations palette) removes an element from the front of a queue and returns the element (see Figure 13.87). If the queue is empty, the function waits timeout in ms before timing out. If an element becomes available in the queue during the wait, the function removes and returns the element and timed out? is FALSE. If queue becomes invalid (for example, the queue reference is released), the function stops waiting and returns error code 1122.

Figure 13.87. Dequeue Element



Flush Queue (Data Communication>> Synchronization>>Queue Operations palette) removes all elements from a queue and returns the elements as an array (see Figure 13.88). This function does not release the queue reference. Use the Release Queue function to release the reference.

Figure 13.88. Flush Queue

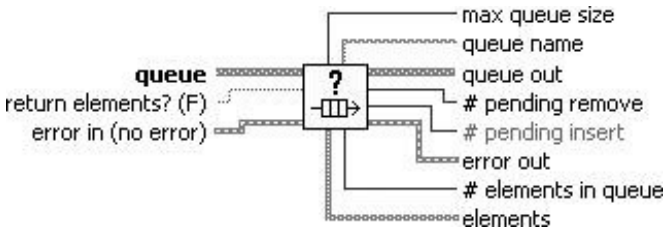


Obtaining Queue Status Information

Get Queue Status (Data Communication>>Synchronization>>Queue Operations palette)

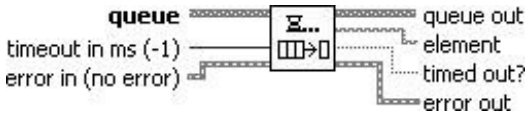
returns information about the current state of a queue, such as the number of elements currently in the queue (see Figure 13.89). You also can use this function to verify that queue is a valid queue refnum. If queue is not a valid queue refnum, the function returns error code 1.

Figure 13.89. Get Queue Status



Preview Queue Element (Data Communication>>Synchronization>>Queue Operations palette) returns the element at the front of the queue without removing the element from the queue (see Figure 13.90). Use the Dequeue Element function to return the element at the front of the queue and remove it from the queue. If the queue is empty, the function waits timeout in ms before timing out. If an element becomes available in the queue during the wait, the function returns the element and timed out? is FALSE. If queue becomes invalid (for example, the queue reference is released), the function stops waiting and returns error code 1122.

Figure 13.90. Preview Queue Element



Activity 13-10: Building a Queued Message Handler Using Queues

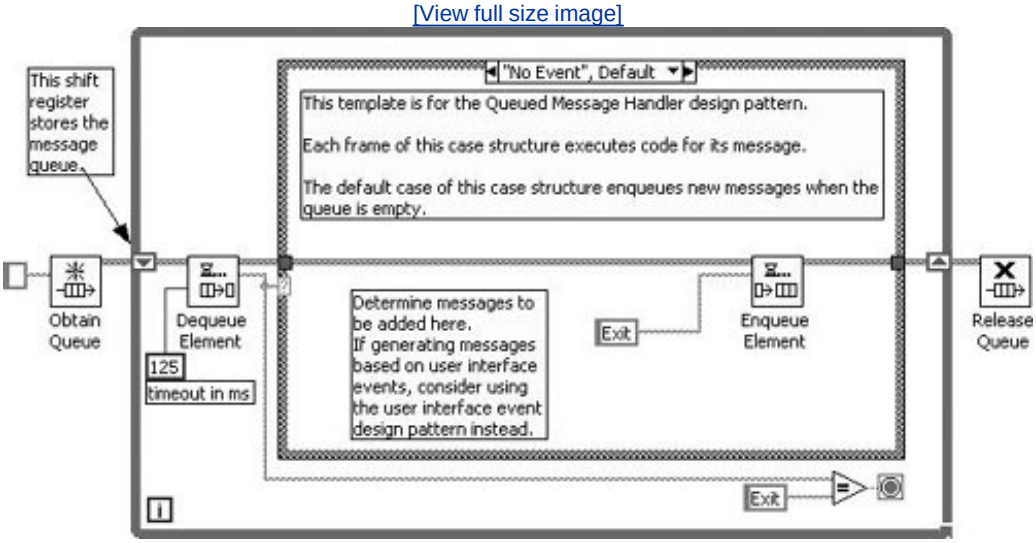
In this activity you modify the Queued Message Handler template to use queues instead of an array of strings.

1. Create a new queued message handler VI, from the template located in the File>>New . . . dialog. From the VI>>From Template>>Frameworks>> Design Patterns node, select Queued Message Handler.

2. Convert the new queued message handler VI, so that it uses a queue, instead of an array of strings, as shown in Figure 13.91.

Figure 13.91. Block diagram of the VI you will create during this activity

[View full size image]



If you do not wire to the timeout input of Dequeue Element (or if you wire a value of 1 into it), it will wait forever and never timeout. If you choose to wait forever, you must ensure that you are able to enqueue a message telling the While Loop to exit, or you can release/destroy the queue (which will cause the Dequeue Element function to stop waiting and return an error).

Note that we have removed the Wait (ms) function from the "No Event" case and have wired a 125 ms timeout to the Dequeue Element function. Because the Dequeue Element will wait 125 ms and then time out (if no elements are in the queue), we no longer need a wait timer inside of the "No Event" case.

3. Save the VI as `Queued Message Handler (using queues).vi`.

Queue Examples, Templates, and Hints

For more examples showing how to use queues, look inside `examples\general\queue.llb`, found beneath your LabVIEW installation. There are also some very good templates that you can use to build your own VIs. Select the File>>New . . . from the menu to launch the New dialog. In the VI>>From Template>>Frameworks>> Design Patterns>> tree node, you will see the following options:

- Producer/Consumer Design Pattern (Events)
- Producer/Consumer Design Pattern (Data)
- Master/Slave Design Pattern

Each of these templates uses queues to implement a very power framework design pattern.

And, here are some final hints for using queues:

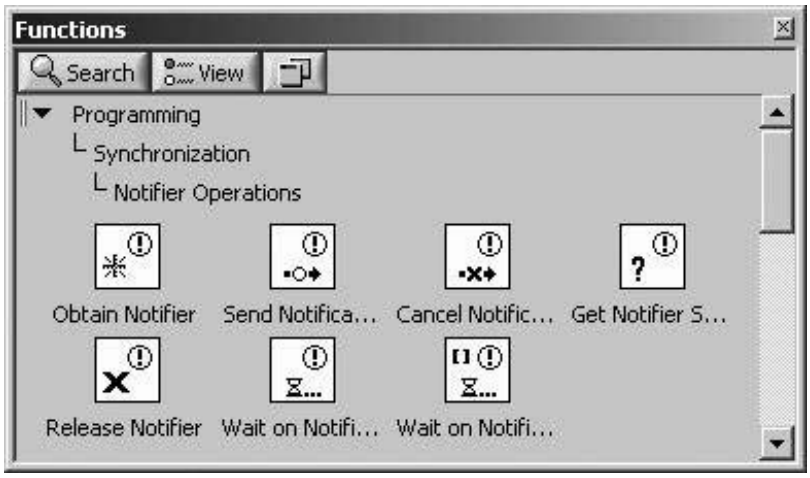
- You should generally only dequeue (read) elements from your queue in one location, called a *consumer*. It is, however, OK (and generally useful) to enqueue (write) elements to your queue in multiple locations, called *producers*. If you feel that you need multiple consumers, you should create a separate queue for each consumer.
- You can obtain a reference to a named queue in multiple locations in your application. Just wire the queue name into the name input of the Obtain Queue function. Don't forget to wire the create if not found input when creating the named queue for the first time, or you will get an error.
- Make sure to call Release Queue for each time you call Obtain Queue. This will help you avoid a memory hog application, which does not clean up after itself. If you are certain that a queue is no longer needed, you can pass a value of TRUE to the force destroy? argument of Release Queuethis will ensure that the queue is destroyed.
- Make your queue element data type a type definition, as described earlier in the section, "Type Definitions." This will allow you to easily change your queue element data type at a later point in time.
- Make your queue reference a type definition, as described in the "Type Definitions" section.
- Consider making your queue element a cluster having two elements: the first element a string called command and the second element a variant called data. This will allow you to pass messages having both commands and arguments (data).

Notifiers

Notifiers are probably the second most commonly used messaging construct (with queues being the most commonly used). The Notifier Operations palette (shown in Figure 13.92) contains all of the functions for operating on notifiers. Just like a queue, a notifier has an element data type that you can define, but it can have only one element (unlike a queue, which has an array of elements). Notifiers are very useful in situations where you only need to have *the latest message, and not all the messages*. The notifier is most useful for broadcasting information to multiple locations, or

consumers. A notifier usually has a single source, or *producer*, but it can (in some instances) be useful to have multiple producers.

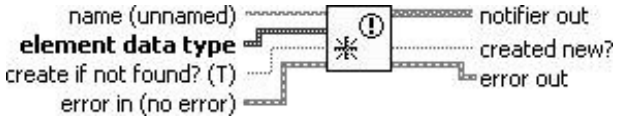
Figure 13.92. Notifier Operations palette



Creating and Destroying Notifiers

Obtain Notifier (Data Communication>>Synchronization>>Notifier Operations palette) returns a reference to a notifier (see Figure 13.93). You must specify the element data type, which is a polymorphic input that accepts any LabVIEW data type. This will define the data type used by the other notifier functions when operating on the resulting notifier.

Figure 13.93. Obtain Notifier

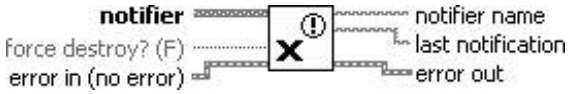


Use named notifiers to pass data between two sections of a block diagram or between two VIs. If you do not wire name, the function creates a new, unnamed notifier reference. If you wire name, the function first searches for an existing notifier with the same name and returns a new reference to the existing notifier. If a notifier with the same name does not already exist and create if not found? is TRUE, the function creates a new, named notifier reference.

Release Notifier (Data Communication>>Synchronization>>Notifier Operations palette) releases a reference to a notifier (see Figure 13.94). You can use the Obtain Notifier function to obtain a reference to the same notifier with the same name multiple times. To destroy a notifier, call

the Release Notifier function a number of times equal to the number of times you obtained a reference to the notifier or stop all VIs using the notifier reference.

Figure 13.94. Release Notifier

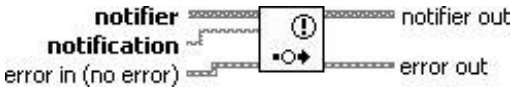


If force destroy? is TRUE, this function releases all references to the notifier and destroys the notifier.

Sending and Waiting on Notification

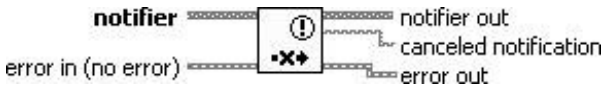
Send Notification (Data Communication>> Synchronization>>Notifier Operations palette) sends a message to all functions waiting on a notifier (see Figure 13.95). All Wait on Notification and Wait on Notification from Multiple functions currently waiting on the notifier stop waiting and continue to execute.

Figure 13.95. Send Notification



Cancel Notification (Data Communication>>Synchronization>>Notifier Operations palette) erases any message currently in a notifier and returns the canceled message (see Figure 13.96). If any Wait on Notification or Wait on Notification from Multiple functions received the message before the call to this function, those functions continue to execute.

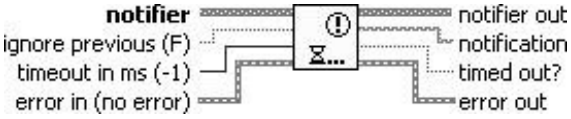
Figure 13.96. Cancel Notification



This function does not recall or reset any wait functions. After you cancel a notification, any subsequent wait functions wait until the notifier receives another message. Canceling a notification before the notifier has a message does not result in an error.

Wait on Notification (Data Communication>>Synchronization>>Notifier Operations palette) waits until a notifier receives a message (see Figure 13.97). When the notifier receives a message, this function continues to execute. Use the Send Notification function to send the message. If a notifier reference becomes invalid (for example, when another function closes it), the function stops waiting and returns error code 1122. If the notifier does not contain a message, this function waits until the notifier receives a message.

Figure 13.97. Wait on Notification



Each unique instance of this function remembers the time stamp of the last message it read.

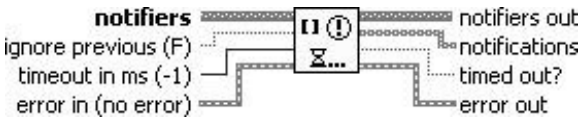
If ignore previous is FALSE, each instance of the Wait on Notification function waits if the message in the notifier has a time stamp for the same time that the instance of the function last executed. If the message is new, then the function returns the message.

When ignore previous is TRUE, the Wait on Notification function always waits for a new message, even if the message currently in the notifier is one it has never seen before.

This function does not remove the message from the notifier. Although a specific instance of the function returns a message only once, other instances of the function or to the Wait on Notification from Multiple function repeat the message until you call the Send Notification function with a new message.

Wait on Notification from Multiple (Data Communication>>Synchronization>> Notifier Operations palette) waits until at least one of the specified notifiers receives a message (see Figure 13.98). When one of the notifiers receives a message, this function continues to execute. Use the Send Notification function to send the message. If a notifier reference becomes invalid, such as when another function closes it, this function stops waiting and returns error code 1122. If the notifier does not contain a message, this function waits until the notifier receives a message.

Figure 13.98. Wait on Notification from Multiple



This function is similar to the Wait on Notification function.

Each unique instance of this function remembers the time stamp of the last message it read. If this

function receives only one message, the function stops remembering which message the time stamp refers to, and the only item filled in the notifiers array is the first element.

If ignore previous is FALSE, each instance of the Wait on Notification from Multiple function determines if one or more notifier has a message newer than the time stamp most recently received by this function. If one or more of the notifiers has new messages, all messages are returned.

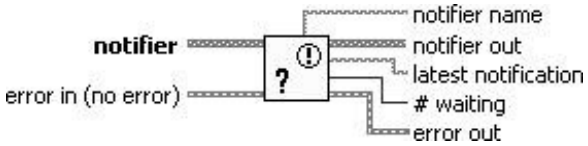
When ignore previous is TRUE, this function always waits for a new message, even if the message currently in the notifier is one it has never seen before.

This function does not remove the message from the notifier. Although a specific instance of the function returns a message only once, other instances of the function or to the Wait on Notification function repeat the message until you call the Send Notification function with a new message.

Obtaining Notifier Status Information

Get Notifier Status (Data Communication>>Synchronization>>Notifier Operations palette) returns information about the current state of a notifier, such as the last uncancelled notification sent to the notifier (see Figure 13.99).

Figure 13.99. Get Notifier Status



Notifier Examples and Hints

For more examples showing how to use notifiers, look inside `examples\general\notifier.llb`, found beneath your LabVIEW installation.

And, here are some final hints for using notifiers:

- You should generally use notifiers only if you are not concerned about missing a notificationyou only care about the latest messages. If you don't want to miss any message, use a queue instead of a notifier.
- Notification can be sent (produced) and received (consumed) in multiple locations. It is generally used to broadcast information, such as status. For example, it is very useful for updating indicators on a user interface, where you only care to update the indicators with the latest data.
- You can obtain a reference to a *named notifier* in multiple locations in your application. Just wire the notifier name into the name input of the Obtain Notifier function. Don't forget to wire the

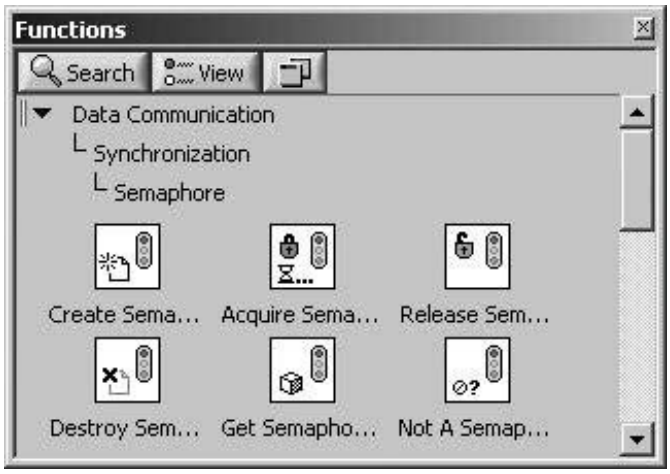
create if not found input when creating the named notifier for the first time, or you will get an error.

- Make sure to call Release Notifier for each time you call Obtain Notifier. This will help you avoid a memory hog application, which does not clean up after itself. If you are certain that a notifier is no longer needed, you can pass a value of TRUE to the force destroy? argument of Release Notifierthis will ensure that the notifier is destroyed.
- Make your notifier element data type a type definition, as described in the earlier section, "Type Definitions." This will allow you to easily change your notifier element data type at a later point in time.
- Make your notifier reference a type definition, as described in an earlier section.
- Consider making your notifier element a cluster having two elements: the first element a string called source and the second element a variant called data. This will allow you to send notification messages containing a lot of useful information.

Semaphores: Locking and Unlocking Shared Resources

The VIs found on the Semaphore palette (shown in Figure 13.100) allow you to lock and unlock a shared resource. A shared resource might be a communication pipeline (such as a TCP connection), an external instrument (such as bench-top GPIB instrument), data (such as a global variables), or anything that can only be operated on by one (or a fixed number of) process(es) at a time.

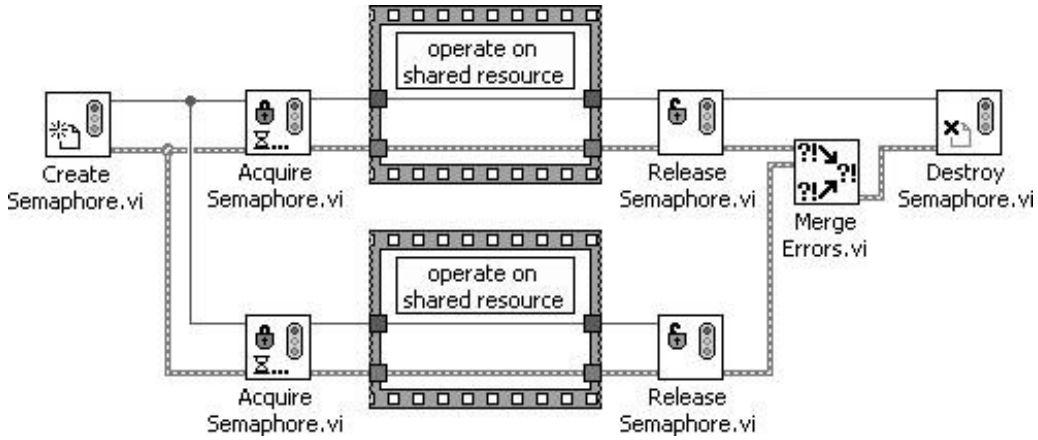
Figure 13.100. Semaphore palette



Unlike a queue or a notifier, *a semaphore has no data*. Rather, it is used purely for synchronizationto provide exclusive sequential access (via a lock/unlock mechanism) to some shared resource. (A semaphore is sometimes referred to as *mutex*, or mutual exclusion.) Figure 13.101 shows a typical use case where a semaphore is used to enforce sequential (one at a time) access to a shared

resource. The secret to how this works is that once Acquire Semaphore (the *lock* function) has been called in one location, if it is called again in any other locations, the subsequent calls will wait until Release Semaphore (the *unlock* function) is called and it is "their turn" to execute. (They will execute in the order that they started waiting.)

Figure 13.101. Semaphore Example.vi showing how to lock a shared resource

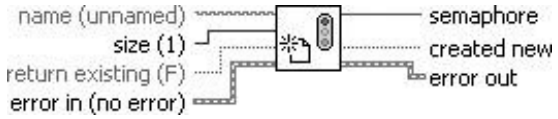


You will learn more about the semaphore functions next.

Creating and Destroying Semaphores

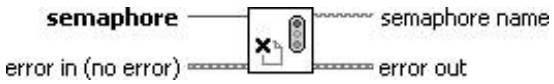
Create Semaphore (Data Communication>>Synchronization>>Semaphore palette) looks up an existing semaphore or creates a new semaphore and returns a refnum (see Figure 13.102). You can use this refnum when calling other Semaphore VIs.

Figure 13.102. Create Semaphore



Destroy Semaphore (Data Communication>>Synchronization>>Semaphore palette) destroys the specified semaphore (see Figure 13.103). All Acquire Semaphore VIs that are currently waiting on this semaphore time out immediately and return an error.

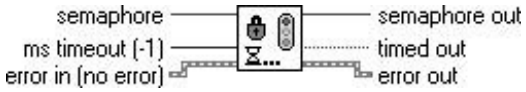
Figure 13.103. Destroy Semaphore



Acquiring (Locking) and Releasing (Unlocking) Semaphores

Acquire Semaphore (Data Communication>>Synchronization>>Semaphore palette) acquires (locks) access to a semaphore (see Figure 13.104).

Figure 13.104. Acquire Semaphore

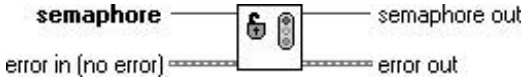


If the semaphore is already acquired by the maximum number of tasks, the VI waits ms timeout milliseconds before timing out. If the semaphore becomes available during the wait, timed out is FALSE. If the semaphore does not become available or semaphore is not valid, timed out is TRUE. The count on a semaphore is incremented each time Acquire Semaphore executes, even if the task acquiring the semaphore has already acquired it once. Acquiring the same semaphore twice without an intervening call to Release Semaphore generally results in incorrect behavior, such as corrupted data values.

This VI attempts to execute even if the error in parameter contains an error.

Release Semaphore (Data Communication>>Synchronization>>Semaphore palette) releases access to a semaphore (see Figure 13.105). If Acquire Semaphore is waiting for the semaphore this VI releases, it stops waiting and continues execution.

Figure 13.105. Release Semaphore



To ensure correct functionality, call Release Semaphore later in the dataflow from Acquire Semaphore. LabVIEW protects the code between Acquire Semaphore and the Release Semaphore, so only as many parallel tasks can run simultaneously with the protected code as the size of the semaphore.

If a semaphore is already acquired and you call Release Semaphore without an Acquire Semaphore

preceding it somewhere earlier in the dataflow, LabVIEW makes the semaphore available for another Acquire Semaphore to proceed, which violates the idea of semaphores protecting shared resources. For most use cases, this results in incorrect behavior, but LabVIEW cannot detect the situation for you.

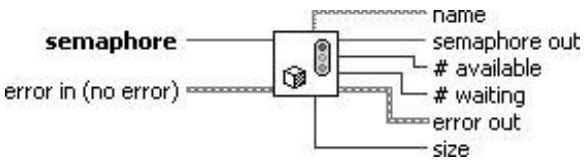
This VI attempts to execute even if the error in parameter contains an error.

If a semaphore is currently unacquired, Release Semaphore returns error code 1111.

Obtaining Semaphore Status Information

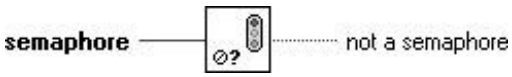
Get Semaphore Status (Data Communication>>Synchronization>>Semaphore palette) returns current status information of a semaphore (see Figure 13.106).

Figure 13.106. Get Semaphore Status



Not A Semaphore (Data Communication>> Synchronization>>Semaphore palette) returns TRUE if semaphore is not a valid semaphore refnum. Note that (unlike with queue and notifier references) you cannot use the Not A Number/Path/Refnum function with a semaphore refnum. You must use the Not a Semaphore function for this purpose (see Figure 13.107).

Figure 13.107. Not A Semaphore



Semaphore Examples and Hints

For more examples showing how to use semaphores, look inside `examples\general\semaphore.llb`, found beneath your LabVIEW installation.

And, here are some final hints for using semaphores:

- Use a semaphore to lock and unlock a shared resource that can only be accessed by one location at a time (or by a fixed number of locations at a time).

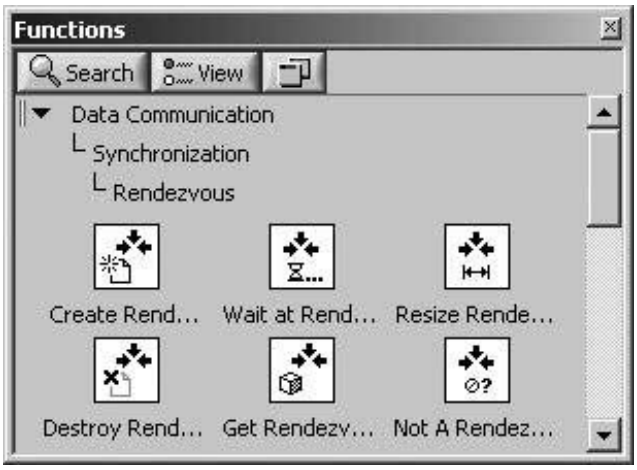
- You cannot use the Not a Number/Path/Refnum function with a semaphore refnum. You must use the Not a Semaphore function for this purpose.
- You can obtain a reference to a *named semaphore* in multiple locations in your application. Just wire the semaphore name into the name input of the Create Semaphore function.
- Make sure to call Destroy Semaphore only once on a semaphore, and only when you are ready to destroy it. Note that each instance of a named semaphore uses the same reference. *If you destroy any instance of a named semaphore, it will destroy it for all users*. This is not the same behavior as queues and notifiers. (Note that queues and notifiers have a Release function, whereas semaphores have a Destroy function. Also, note that *the Release Semaphore function does not release the reference, but rather releases the semaphore/lock*.)

What would happen if semaphores did have data? What if the Acquire Semaphore function read the data and the Release Semaphore function wrote the data? This is exactly what the traditional Graphical Object-Oriented Programming (GOOP) framework does. You can learn more about GOOP and Object-Oriented Programming in Appendix D, "LabVIEW Object-Oriented Programming."

Rendezvous

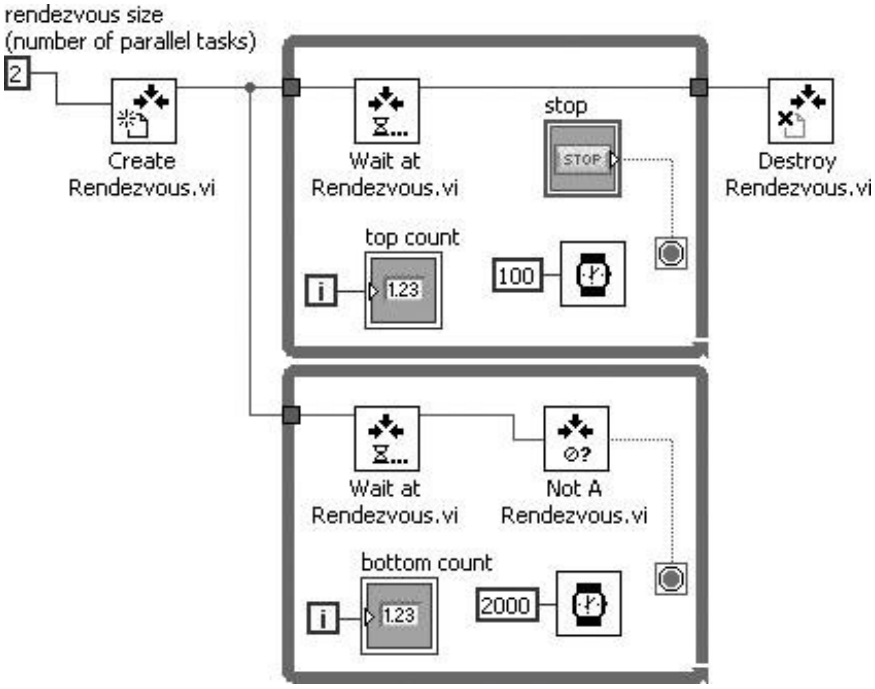
The VIs found on the Rendezvous palette (shown in Figure 13.108) allow you to synchronize two or more separate, parallel tasks at specific points of execution. Each task that reaches the rendezvous waits until the specified number of tasks are waiting, at which point all tasks proceed with execution.

Figure 13.108. Rendezvous palette



Unlike a queue or a notifier, *a rendezvous has no data*. Rather, it is used purely for synchronizationto cause parallel tasks to all wait at a specific point before proceeding. Figure 13.109 shows a typical use case where a rendezvous is used to synchronize parallel tasks in separate loops. When this code executes, *the loops will run at the rate of the slowest loop*, due to the presence of the Wait at Rendezvous function.

Figure 13.109. Rendezvous Example.vi showing how to synchronize parallel tasks

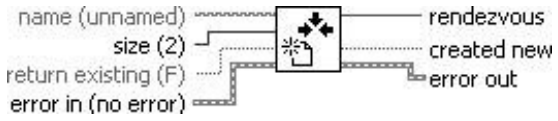


You will learn more about the rendezvous functions next.

Creating and Destroying Rendezvous

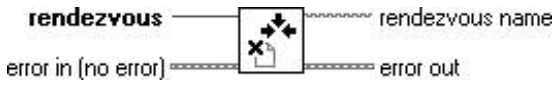
Create Rendezvous (Data Communication>>Synchronization>>Rendezvous palette) looks up an existing rendezvous or creates a new rendezvous and returns a refnum (see Figure 13.110). You can use this refnum when calling other Rendezvous VIs.

Figure 13.110. Create Rendezvous



Destroy Rendezvous (Data Communication>>Synchronization>>Rendezvous palette) destroys the specified rendezvous (see Figure 13.111). All Wait at Rendezvous VIs that are currently waiting on this rendezvous time out immediately and return an error.

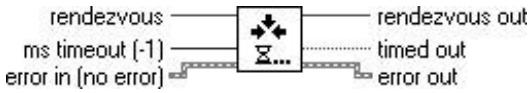
Figure 13.111. Destroy Rendezvous



Waiting on a Rendezvous

Wait at Rendezvous (Data Communication>> Synchronization>>Rendezvous palette) waits until a sufficient number of tasks have arrived at the rendezvous (see Figure 13.112).

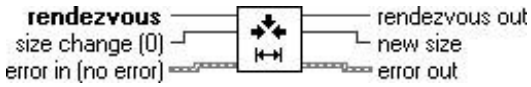
Figure 13.112. Wait at Rendezvous



Resizing a Rendezvous

Resize Rendezvous (Data Communication >>Synchronization>>Rendezvous palette) changes the size of rendezvous by size change and returns new size (see Figure 13.113). Remember, the size of a rendezvous is the number of instances of Wait at Rendezvous that must be waiting before all instances can proceed.

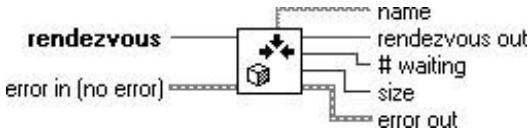
Figure 13.113. Resize Rendezvous



Obtaining Rendezvous Status Information

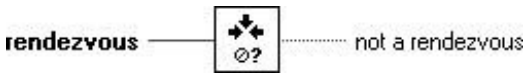
Get Rendezvous Status (Data Communication>>Synchronization>>Rendezvous palette) returns current status information of a rendezvous (see Figure 13.114).

Figure 13.114. Get Rendezvous Status



Not A Rendezvous (Data Communication>>Synchronization>>Rendezvous palette) returns TRUE if rendezvous is not a valid rendezvous refnum. Note that (unlike with queue and notifier references) you cannot use the Not a Number/Path/Refnum function with a rendezvous refnum. You must use the Not a Rendezvous function for this purpose (see Figure 13.115).

Figure 13.115. Not A Rendezvous



Rendezvous Examples and Hints

For more examples showing how to use rendezvous, look inside `examples\general\rendezvous.llb`, found beneath your LabVIEW installation.

And, here are some final hints for using rendezvous:

- Use a rendezvous to synchronize two or more separate, parallel tasks at specific points of execution. Each task that reaches the rendezvous waits until the specified number of tasks are waiting, at which point all tasks proceed with execution.
- Make sure that rendezvous size is equal to the actual number of parallel tasks that you want to synchronize. Otherwise, you might not wait for all of the tasks to complete because you set the size to a value less than the number of parallel tasks, or you might wait forever because you set the size to a value larger than the number of parallel tasks.
- You cannot use the Not a Number/Path/Refnum function with a rendezvous refnum. You must use the Not a Rendezvous function for this purpose.
- You can obtain a reference to a *named rendezvous* in multiple locations in your application. Just wire the rendezvous name into the name input of the Create Rendezvous function.
- Make sure to call Destroy Rendezvous only once on a rendezvous, and only when you are ready to destroy it. Note that each instance of a named rendezvous uses the same reference. *If you destroy any instance of a named rendezvous, it will destroy it for all users*. This is not the same behavior as queues and notifiers. (Note that queues and notifiers have a Release function, whereas rendezvous have a Destroy function.)

Occurrences

The VIs found on the Occurrences palette (shown in Figure 13.116) allow you to control separate, synchronous activities, in particular, when you want one VI or part of a block diagram to wait until another VI or part of a block diagram finishes a task without forcing LabVIEW to poll.

Figure 13.116. Occurrences palette

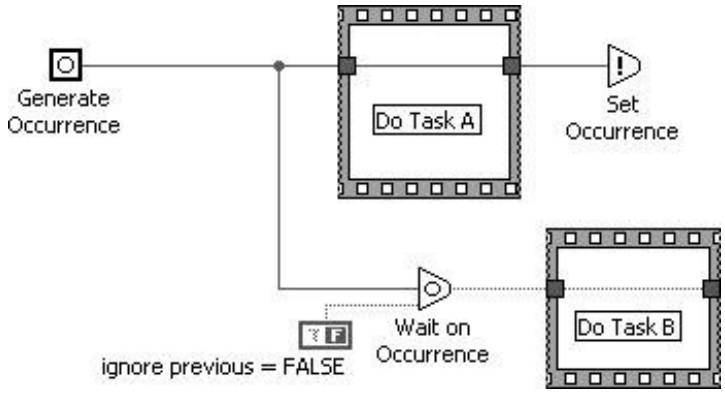
You can perform the same task using global variables (or local variables within the same VI), with one loop polling the global variable until its value changes. However, global variables use more overhead because the loop that waits uses execution time. With occurrences, the second loop becomes idle and does not use processor time. When the first loop sets the occurrence, LabVIEW

activates the second loop and any other block diagrams that wait for the specified occurrence.

You are encouraged to use notifiers instead of occurrences for most operations. However, for some memory- and processor-intensive event-driven programs, occurrence might be the best choice.

Unlike the other synchronization VIs, *there is no destroy function for occurrences*each instance of the Generate Occurrence function acts similarly to a constant, outputting the same value each time it is called. Additionally, *occurrences have no data*they are used purely for synchronization where you want to wait until an event occurs in some other location (or the wait function times out). Figure 13.117 shows Occurrence Example.vi (located in the `EVERYONE\CH13` folder on the CD-ROM), an example where an occurrence is used synchronize two parallel tasks. In this example, Task B cannot execute until Task A completes, causing Set Occurrence to execute and therefore cause Wait on Occurrence (which was waiting for the occurrence to be set) to stop waiting.

Figure 13.117. Occurrence Example.vi showing how to synchronize parallel tasks



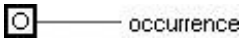
You will learn more about the occurrence functions next.

Creating Occurrences

Generate Occurrence (Data Communication>>Synchronization>>Occurrences palette) creates an occurrence that you can pass to the Wait on Occurrence and Set Occurrence

functions (see Figure 13.118).

Figure 13.118. Generate Occurrence



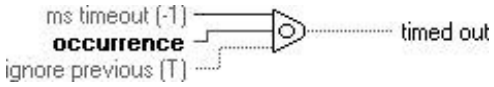
Ordinarily, only one Generate Occurrence function is wired to any set of Wait on Occurrence and Set Occurrence functions. You can wire a Generate Occurrence function to any number of Wait on Occurrence and Set Occurrence functions. You do not have to have the same number of Wait on Occurrence and Set Occurrence functions.

Unlike other synchronization VIs (such as queues, notifiers, etc.), each Generate Occurrence function on a block diagram represents a single, unique occurrence. In this way, the Generate Occurrence function is similar to a constant. When a VI is running, every time a Generate Occurrence function executes, the function produces the same value. For example, if you place a Generate Occurrence function inside of a loop, the value produced by the Generate Occurrence function is the same for every iteration of the loop. If you place a Generate Occurrence function on the block diagram of a reentrant VI, the Generate Occurrence function produces a different value for each caller.

Waiting on and Setting Occurrences

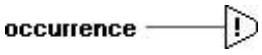
Wait on Occurrence (Data Communication>>Synchronization>>Occurrences palette) waits for the Set Occurrence function to set or trigger the given occurrence (see Figure 13.119).

Figure 13.119. Wait on Occurrence



Set Occurrence (Data Communication>>Synchronization>>Occurrences palette) triggers the specified occurrence. All nodes that are waiting for this occurrence stop waiting (see Figure 13.120).

Figure 13.120. Set Occurrence



Occurrence Examples and Hints

For more examples showing how to use occurrences, look inside `examples\general\occurrence.llb`, found beneath your LabVIEW installation.

And, here are some final hints for using occurrences:

- Generally, you should use occurrences only for memory and performance optimization. Try using notifiers instead of occurrences.
- Occurrences don't have a destroy function (like other synchronization functions). Each Generate Occurrence function acts as a constant, outputting the same occurrence reference each time it is called.

First Call?

The First Call? function (see Figure 13.121) is found on the Data Communication>>Synchronization palette. The Boolean output of this function indicates that a subVI or section of a block diagram is running for the first time.

Figure 13.121. First Call?

The First Call? function (Data Communication>>Synchronization palette) returns TRUE only the first time you call it after you click the Run button. You can place the First Call? function in multiple locations within a VI. The function returns TRUE the first time the section of the block diagram in which it is placed runs.

This function is useful if you want to run a subVI or a section of a block diagram within a loop or Case Structure only once when the VI runs.

First Call? returns TRUE the first time the VI runs after the first top-level caller starts running, such as when the Run button is clicked or the Run VI method executes. If a second top-level caller calls the VI while the first top-level caller is still running, First Call? does not return TRUE a second time. After all the top-level callers become idle and a top-level caller starts again, First Call? returns TRUE the first time the VI runs after the idle state. Reentrant VIs (discussed in Chapter 15) have an instance per data space. Therefore, a shared reentrant VI returns TRUE for each data space instance the first time its top-level caller calls it.

PREV NEXT

Structures for Disabling Code

If you have programmed in a text-based language, you have probably commented out blocks of code. For example, in C and Java, you can use the comment start (/*) and comment end (*/) strings to define a range of text that should not be executed. You can also use the line comment (//) string for commenting out entire lines of code. Shown next is a small snippet of code that uses both of these commenting techniques:

```
/*
* This is an example showing
* how to comment out text code
*/

if (foo > 1) {

 // System.out.println("Hello World!");

}
```

In LabVIEW, floating text placed on the block diagram does affect the functionality of the VIbecause LabVIEW is not a text-based language, floating text is ignored. However, for specifying portions of your graphical code that you do not want LabVIEW to execute, you can use the Diagram Disable and Conditional Disable structures. These can both be found in the Programming>>Structures subpalette of the Functions palette.

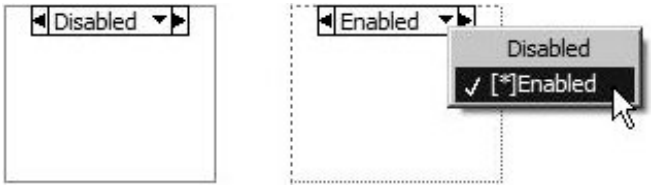
Just like a Case Structure, the Diagram Disable and Conditional Disable structures allow you to create several subdiagrams. However, you will notice that unlike the Case Structure, the Diagram Disable and Conditional Disable structures have no selector terminals for choosing which subdiagram LabVIEW will execute at run-time. This is because the one frame that LabVIEW is going to run is defined at edit-time. We will now discuss the way that the Diagram Disable and Conditional Disable structures decide which frame gets to run.

The Diagram Disable Structure

With the Diagram Disable structure, we can create multiple frames and name them anything we wish. And, we can define (at most) one frame that is enabled. You can enable a subdiagram by

making the subdiagram visible and then right-clicking on the frame of the Diagram Disable structure and choosing *Enable This Subdiagram* from the shortcut menu. Similarly, you can disable a subdiagram by choosing *Disable This Subdiagram* from the shortcut menu. When you enable a subdiagram, all other subdiagrams will be disabled. Only one subdiagram can be enabled at a time. However, you do not have to have any subdiagrams enabled. If no subdiagram is enabled, then none of the subdiagrams will execute. You can tell which subdiagram (if any) is enabled from the pop-up menu on the subdiagram selectorthe subdiagram that is enabled will begin with an asterisk in square brackets ("[*]"), as shown in Figure 13.122.

Figure 13.122. Diagram Disable structure



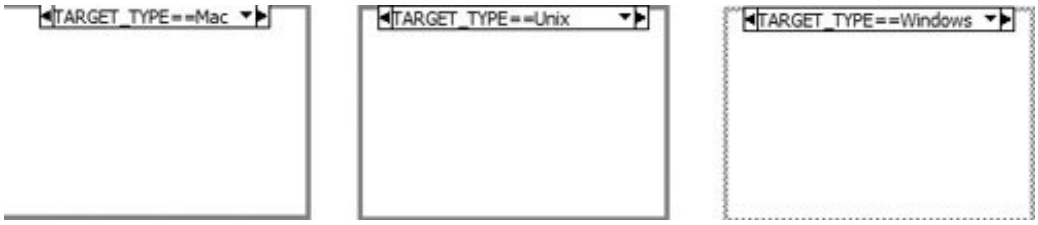
By default, LabVIEW names the disabled subdiagrams "Disabled" and the enabled subdiagrams "Enabled." You can have several frames named "Disabled." Similarly, you can name several subdiagrams with the same nameunlike a Case Structure, LabVIEW will not complain about duplicate Diagram Disabled subdiagram names. If you do not change the subdiagram's name, LabVIEW will rename it when you change it from enabled to disabled and vice versa. However, LabVIEW will not rename any frames whose name you have changed. Remember, you can always tell which frame, if any, is enabled by the presence of "[*]" before its name.

Disabling code does not just prevent it from executing; *LabVIEW doesn't even compile code in the "Disabled" subdiagrams*. The usefulness of this is that any code in a "Disabled" subdiagram that is broken (or cannot be found) does not break our VI.

The Conditional Disable Structure

The Conditional Disable structure is slightly different from the Diagram Disable structure. The first difference is that LabVIEW will execute *exactly one* of the Conditional Disable structure subdiagramsthis subdiagram is referred to as the *active subdiagram* and the other subdiagrams are referred to as *inactive subdiagrams*. The second difference is that LabVIEW evaluates a *condition* (a conditional expression) to determine which frame will executeeach subdiagram's condition is shown as the title of that subdiagram, as shown in Figure 13.123.

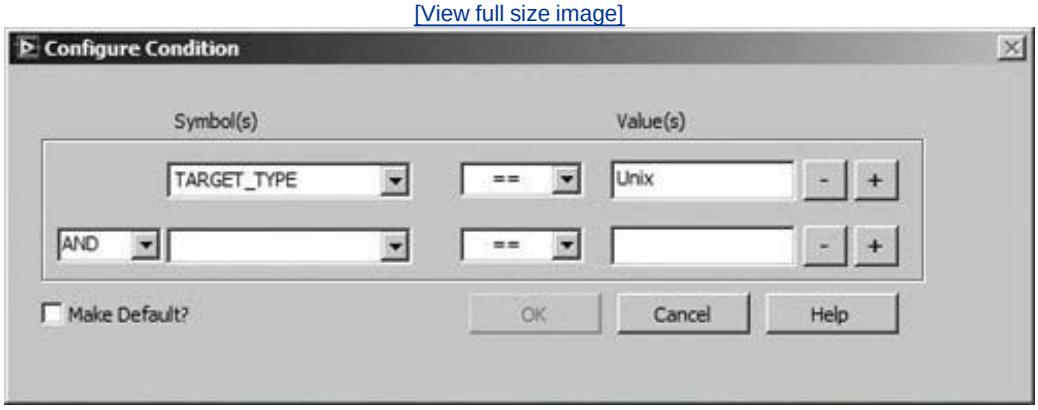
Figure 13.123. Conditional Disable structure



A subdiagram's condition may be edited by selecting the subdiagram, and then right-clicking on the frame of the Conditional Disable structure and selecting *Edit Condition for This Subdiagram* from the shortcut menu. This will open the Configure Condition dialog, shown in Figure 13.124. From this dialog, you can edit the subdiagram's conditional expression.

Figure 13.124. Configure Condition dialog

[View full size image]



For example, you can define one subdiagram with the condition "TARGET_ TYPE= =Windows" and another subdiagram with the condition "TARGET_TYPE = =Mac," in order to have a VI that executes differently on Windows than it does on the Mac. However, what LabVIEW does is more interesting than just deciding which code will execute on a given platform. *LabVIEW only compiles the code in the active subdiagram* (the subdiagram whose Condition evaluates to TRUE). The usefulness of this is that any code in an inactive subdiagram that is broken (or cannot be found) does not break our VI.

You might have noticed that the only symbol available is TARGET_TYPE. You can define

other symbols in the LabVIEW Project environment; however, we will leave this as an exercise for the adventurous reader.

PREV NEXT

Halting VI and Application Execution

LabVIEW gives you a couple of functions to abort the execution of your code immediately: Stop and Quit LabVIEW, which are found on the Programming>>Application Control palette.

Stop Function

The Stop function has a Boolean input. When TRUE (which is the unwired default input), it halts execution of the VI and all its subVIs, just as if you had pressed the stop button on the Toolbar.

Quit LabVIEW Function

The Quit LabVIEW function does just that, when its input is TRUE (also the default). Be careful with this one.

Some of us reminisce about a third function that was included back when LabVIEW was in version 2.2: Shutdown. On certain Macintosh models, this function would effectively turn off the whole computer, monitor and all. We never understood how it could be useful, but it was fun to leave a colorful VI running with one big button that said "DO NOT PRESS THIS BUTTON" on someone else's Mac, and then just wait to see who would be too curious . . . . The button was, of course, connected to the operating system's Shutdown function. It was a good way to amuse ourselves at the expense of the less-experienced LabVIEW users.

On a serious note, however, you should use these functions with caution. It is considered bad programming etiquette (in any language) to include a "hard stop." Strive to always have a graceful way to exit your program.

Make sure your VI completes all final tasks (closing files, finishing the execution of subVIs, etc.) before you call the Stop function. Otherwise, you may encounter unpredictable file behavior or I/O errors, and your results can be corrupted.

Cool GUI Stuff: Look What I Can Do!

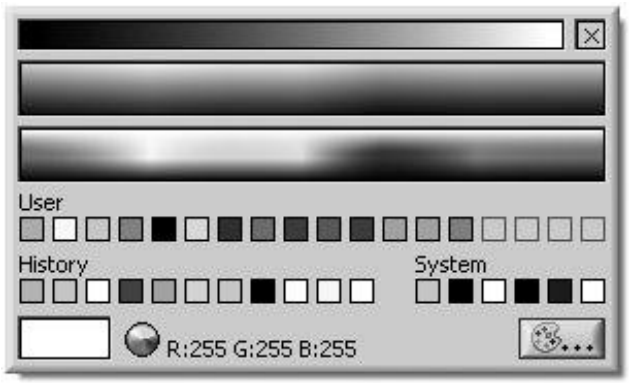
LabVIEW provides some powerful GUI components that you can use to make your application look very slick! Why should using software be boring? Let's take a look at some graphical user interface components that will add a certain "wow!" factor to your applications.

System Controls and Colors

System controls (called dialog controls prior to LabVIEW 8) and system colors change with the operating system and desktop theme/style a user has selected. This is a very useful feature for several reasons. First, it makes your application look and feel like "real software." Large companies like Microsoft, Apple, and IBM spend millions of dollars figuring out what looks good, so why not leverage off their investmentthese companies even publish user interface standards based on their research. Second, because the dialog controls and colors change depending on the theme the user has selected, your application will respond to those changesthis is a bonus feature of your application that you get for free! It is important to realize that sometimes users select specific desktop themes for reasons other than what they think looks good or appealing. Some users have special eye-sight or other physical needs that are accommodated by large fonts, high contrast, or specific color combinations. Giving users control over how your software appears opens up a large market of users who might not otherwise be able to use your software. System controls and colors may help you meet "accessibility standards."

System colors are found on the color dialog (see Figure 13.125), in the lower-right corner. The word "System" is directly above this row of colors.

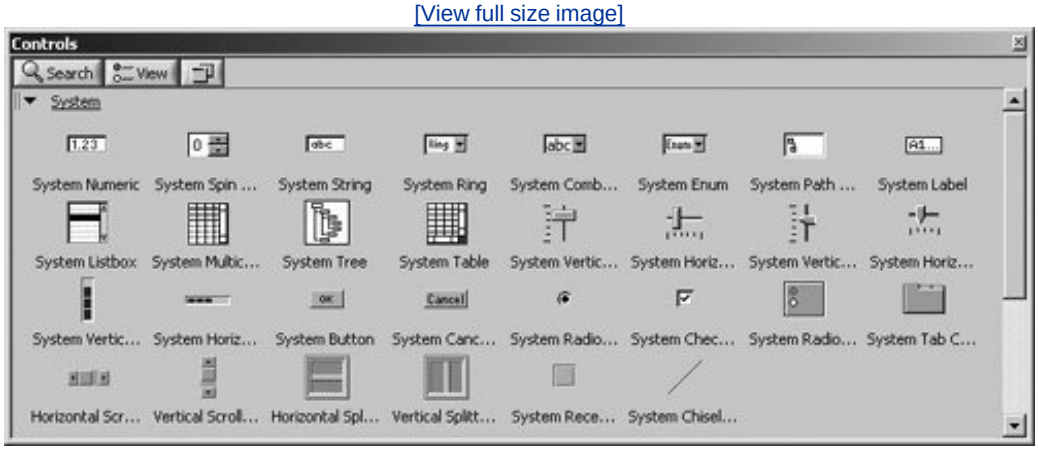
Figure 13.125. Color dialog, showing System colors in the lower-right



System controls are found on the System subpalette of the Controls palette (see Figure 13.126).

Figure 13.126. System palette

[View full size image]



Figures 13.127 through 13.129 show examples of how system controls appear on the different platforms.

Figure 13.127. System controls on Mac OS X

[View full size image]

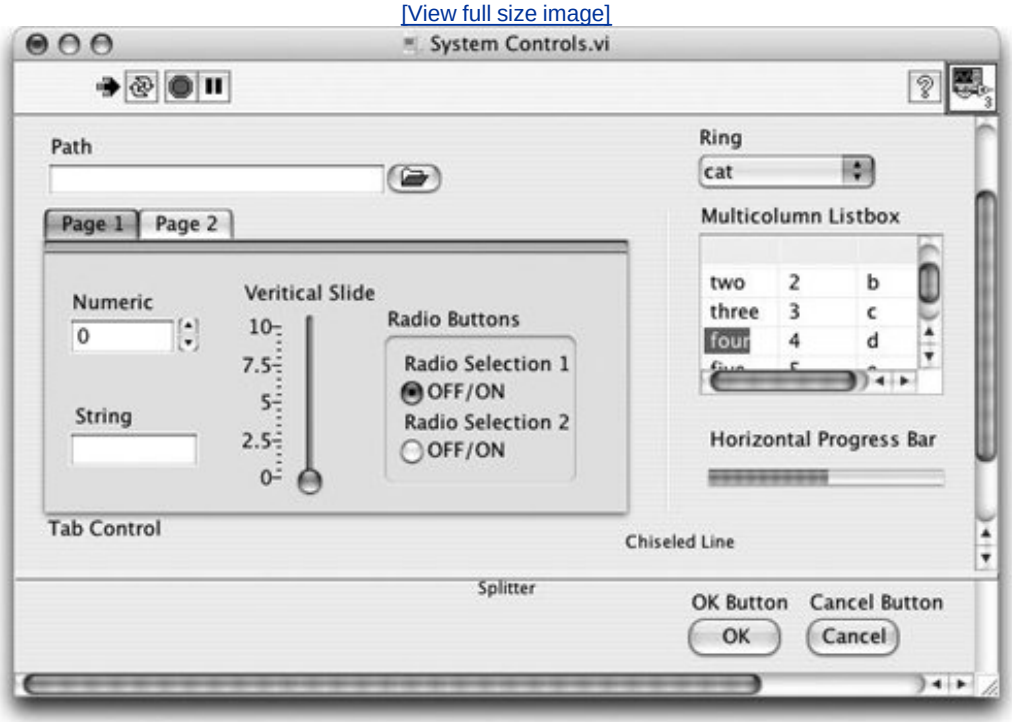


Figure 13.128. System controls on Windows XP (XP Silver Theme)

[View full size image]

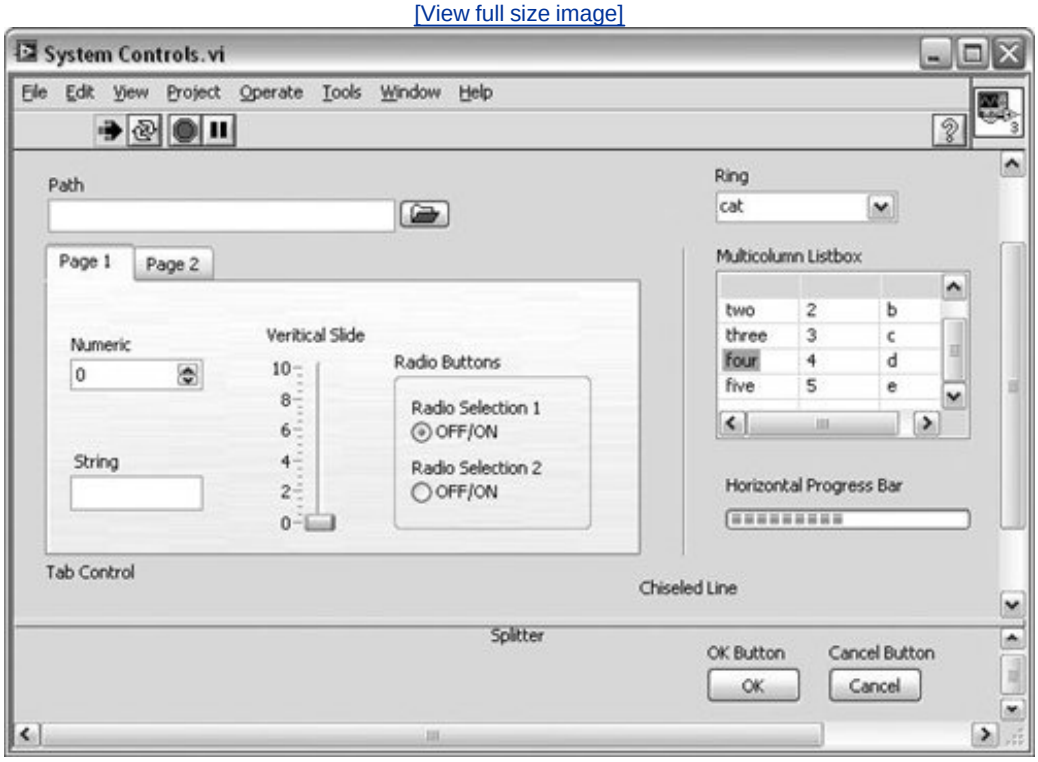
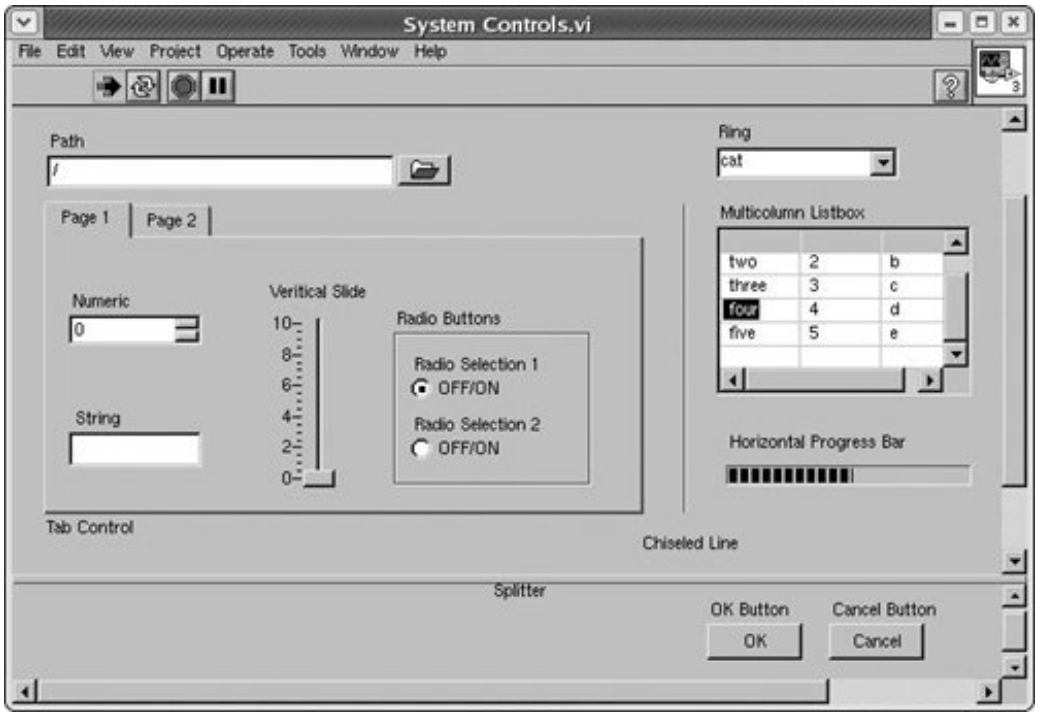


Figure 13.129. System controls on Linux

[View full size image]



Make sure to open your VIs and inspect the front panels on any platforms on which you anticipate your software might be used. The sizes of fonts and controls can vary slightly, so you will want to make sure that there are not problems with overlapping text or controls. We will discuss more cross-platform considerations in Chapter 17.

Drag and Drop

Drag and drop is a very exciting feature in LabVIEW. It is very powerful, allowing you to drag and drop data to and from *any* control or indicatoreven between different VIs! This is a very advanced topic, and we will not go into great detail on how everything works, but we will give you an overview of the key components.

There are several components of this feature:

1. The control Start Drag method (see Figure 13.130). This method starts a drag and drop operation using the control as the source. You get to specify Data, which is an array of clusters containing data name (a string) and drag data (a variant, which can be anything you wish). The data names are very important; you will use the Get Drag Drop Data function (item #3 in this list) to access the drag data by name.

Figure 13.130. Start Drag method

[View full size image]

2. The Drag and Drop events, available for controls via the Event Structure. There are several drag and drop events that are available for all controls:

   - Drag Ended
   - Drag Enter
   - Drag Leave
   - Drag Over
   - Drag Source Update
   - Drop

   Each of these has an event data element called Available Data Names, which is an array of strings containing the data names that you defined when you called the Start Drag method.

   Some of these events also have event filter data, allowing you to do things such as rejecting a drop event, for example.

3. The Get Drag Drop Data function, shown in Figure 13.131, is found on the Programming>>Application Control palette. This function returns drag data from LabVIEW when you perform a drag and drop operation. Only use this function when it is necessary to access the drag data, not just to examine the data type. If a drag and drop operation is not in progress, LabVIEW returns an error. If the data requested is unavailable, LabVIEW returns an error.

Figure 13.131. Get Drag Drop Data function

For some excellent drag and drop LabVIEW examples, look in the `examples\general\dragdrop` folder in your LabVIEW installation.

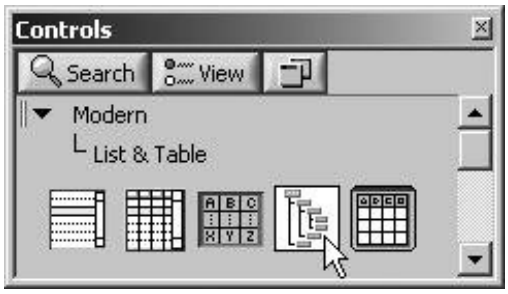
Tree Control

The tree control may be one of the best user interface widgets ever invented. It displays a hierarchical list of items that the user can interact with by selecting items, dragging and dropping items, expanding and contracting items, etc. You've already experienced tree controls when using NI Example Finder, Measurement & Automation Explorer (MAX), the LabVIEW Help documentation, and several other applications on your computer. Using tree controls is very easy; programming tree controls is a little trickier. It's hard work making things so easy for your application's users; but somebody has to do it, so let's get started!

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

You can find the Tree Control on the Modern>>List & Table palette (see Figure 13.132).

Figure 13.132. List & Table palette



Place a tree on the front panel of a VI and you will notice that it appears similar to a multi-column listbox (see Figure 13.133) and its terminal is a string data type (see Figure 13.134). The value of that string is the *tag* of currently selected item. Before we define what a tag is, let's add some items to our tree control.

Figure 13.133. Tree control on the front panel

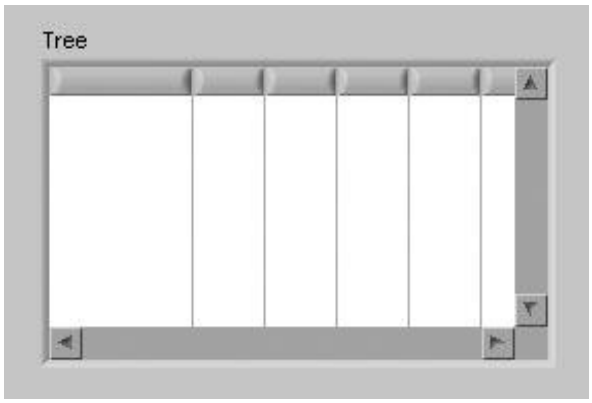
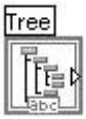


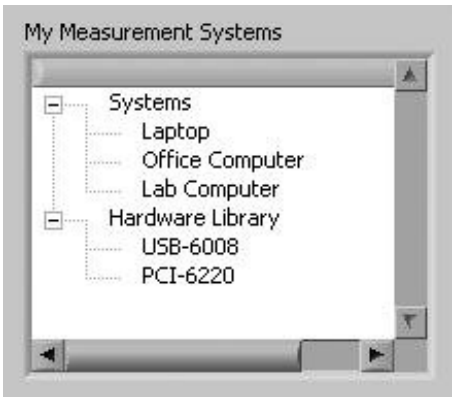
Figure 13.134. Tree control terminal on the block diagram



Using the Labeling tool, enter an item name in the first column of the first row of the tree control and press the <Enter> key. You have just created your first item in the tree! And, when you created the item, LabVIEW also created a *tag* for the itemthe tag is the text that you entered, unless an item already had that tag (in which case, LabVIEW will append a number, in order to make the tag unique). You can't see the tag, but it is important for programmatically controlling the tree and interpreting user actions with the tree's items.

Now create a few more items. All of the items in your tree are at the same level, but you can create a *hierarchy* (tree) by indenting items, which makes them child-nodes of the items above them. To indent (or outdent) an item, right-click on the item and select Indent Item (or Outdent Item) from the shortcut menu. You should now have a hierarchy of items, similar to the one shown in Figure 13.135.

Figure 13.135. Tree control showing a hierarchy of items



Another way to create a hierarchy is to drag and drop items. Using the Operating tool, you can drag an item onto another item to make the first item a child of the second item. For example, you might wish to move the "USB-6008" DAQ device from the "Hardware Library" into the "Laptop" system, as shown in Figures 13.136 and 13.137.

Figure 13.136. Drag and drop (before)

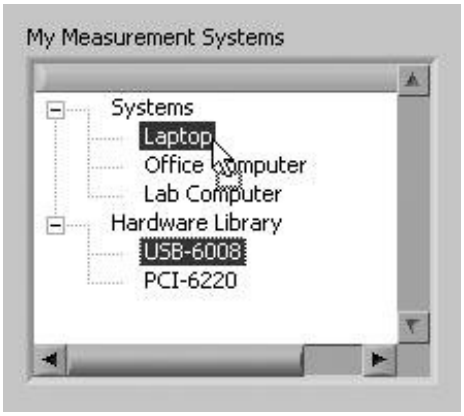
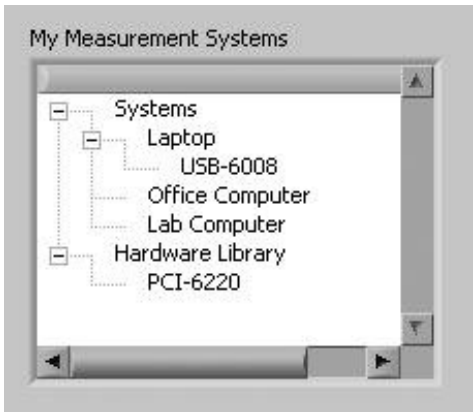


Figure 13.137. Drag and drop (after)



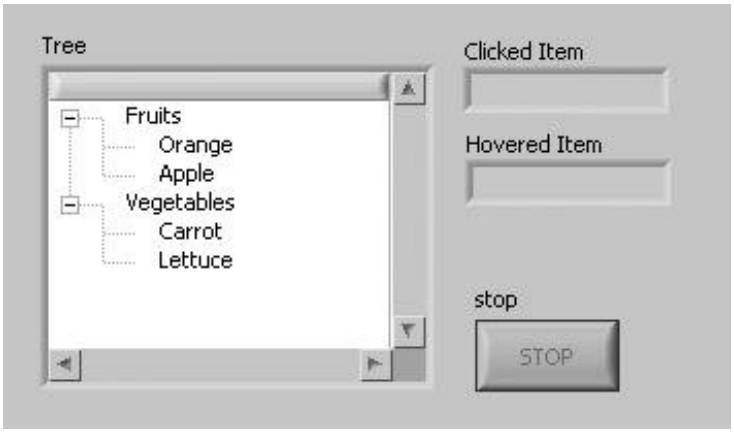
You can allow users to edit the cell string values of a tree control at run-time. Pop up on a tree control and select Editable Cells from the shortcut menu to enable this feature. You also can use the Allow Editing Cells property (using a property node) to programmatically allow users to edit cells in a single-column listbox, multicolumn tree control. Use the Edit Cell event (using an Event Structure) to handle the event that occurs when a user changes the text.

Activity 13-11: Capturing Mouse Events on a Tree Control

In this activity you will build a VI that captures mouse events on a tree control.

1. Build a front panel with a tree control, as shown in Figure 13.138.

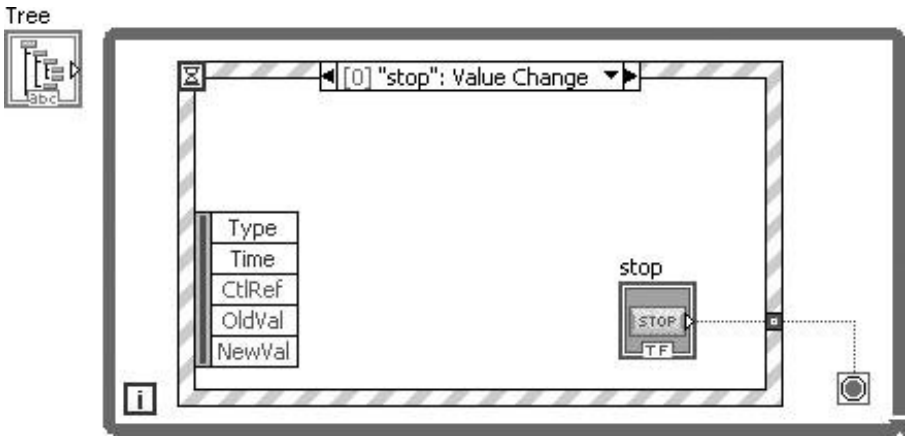
Figure 13.138. Front panel of the VI you will create during this activity



2. Use the Labeling tool to add elements to the tree and use the Operating tool to drag elements so that they become child-nodes of other elements.

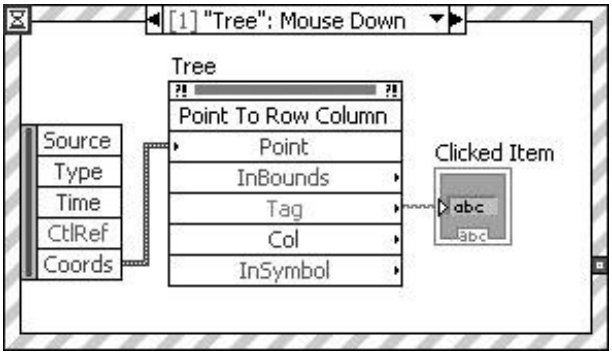
3. Build the block diagram so that it has a While Loop with an Event Structure, as shown in Figure 13.139. Add an event for the `stop` button's Value Changed event. Place the `stop` button terminal inside this event case and wire it to the While Loop's Conditional Terminal, which should be configured as stop if true.

Figure 13.139. Your VI's Event Structure after you add the "stop": Value Change event



4. Add an event for the `TRee` control's Mouse Down event, as shown in Figure 13.140. Create an Invoke Node that calls the `TRee` control's Point To Row Column method, and wire the Coords event data into the Point argument of the methodthis will convert the mouse coordinates into information about where the mouse is currently located on the tree control.

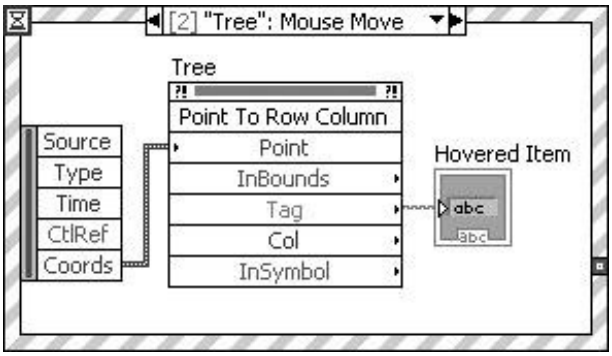
Figure 13.140. Your VI's Event Structure after you add the "Tree": Mouse Down event



The Point To Row Column method converts a pixel coordinate to a tag-column pair in the coordinates of the control. This method also returns whether the point is inside the bounds of the content rectangle and whether the point is within the custom symbol of the cell. Wire the Tag output to the `Clicked Item` indicator terminal.

5. Add an event for the `tree` control's Mouse Move event, as shown in Figure 13.141. Build this case, in the same way as the Mouse Down case. Wire the Tag output to the `Hovered Item` indicator terminal.

Figure 13.141. Your VI's event structure after you add the "Tree": Mouse Move event



6. Save your VI as `tree Control Mouse Events.vi`.

If you use the Event Structure's Duplicate Event Case . . . pop-up option, instead of Add Event Case . . ., to create the new event case from the Mouse Down case, the

`Clicked Item` *indicator (which is inside the Mouse Down case) will be duplicated. Sometimes you will want to use this technique to save some editing steps. For example, you could rename it* `Hovered Item`.

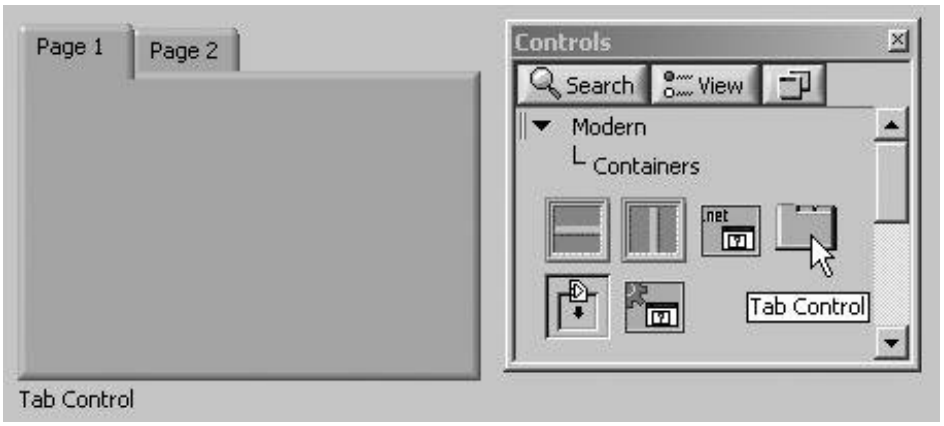
7. Run the VI. Notice how the `Hovered Item` indicator displays the item that the mouse is hovering over and how the `Clicked Item` indicator displays the item that the user clicks with the mouse.

This example just scratches the surface of what the tree control can do. For more exciting examples, look in the LabVIEW `examples\general\controls\treecontrol` folder.

Tab Control

The tab control (found on the Modern>>Containers subpalette of the Controls palette) is a very useful control that can be used to both group and hide other front panel controls (see Figure 13.142).

Figure 13.142. Placing a Tab Control onto a VI's front panel from the Modern>>Containers palette



You can place any number of controls and indicators on each page of the tab control, as shown in Figures 13.143 and 13.144. The user can mouse-click on the page that he would like to make visible.

Figure 13.143. Controls and indicators on the first page of a tab control

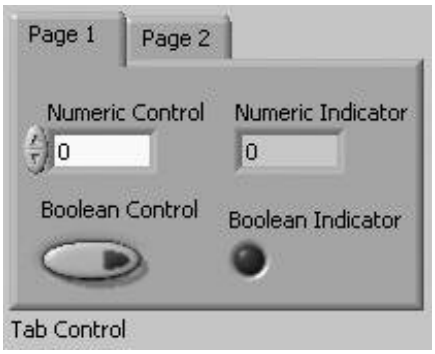
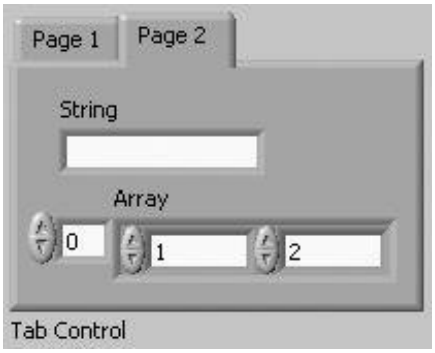
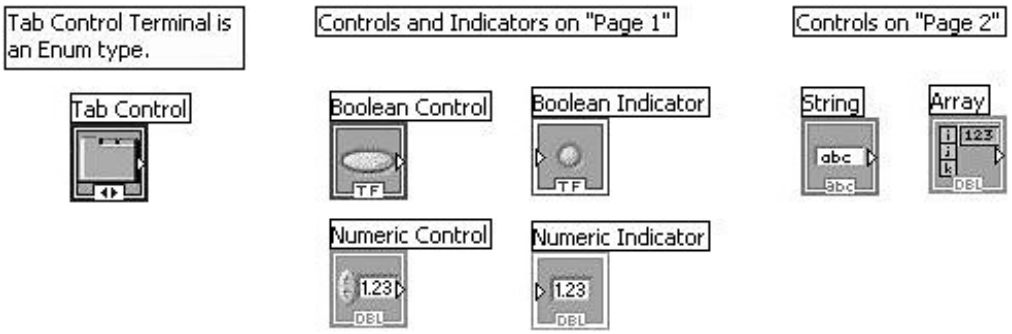


Figure 13.144. Controls and indicators on the second page of a tab control



If we look at the block diagram, we will see that, although the front panel controls and indicators are grouped, *the block diagram terminals are not grouped* (see Figure 13.145).

Figure 13.145. Block diagram containing the terminal of a tab control, as well as the terminals of the controls and indicators contained on the pages of the tab control



A tab control is not like a cluster; it does not bundle the controls together into its data. Rather, a tab control is an enum, with the possible values being the labels (names) of its pages. To see this, pop up on the tab control's block diagram terminal and select Create Constant, and then inspect its possible values (as shown in Figures 13.146 and 13.147).

Figure 13.146. Tab control terminal and an enum constant created by selecting Create Constant from the tab control's pop-up menu

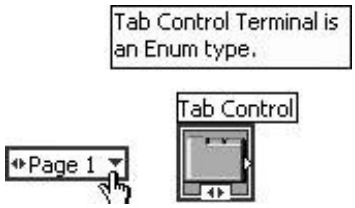
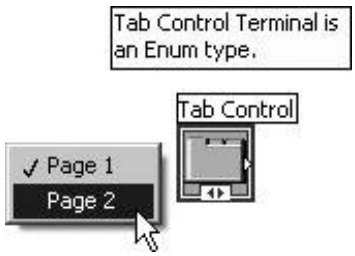


Figure 13.147. Viewing the item names from the pop-up menu of the enum constant to show that it contains items for each page of the tab control



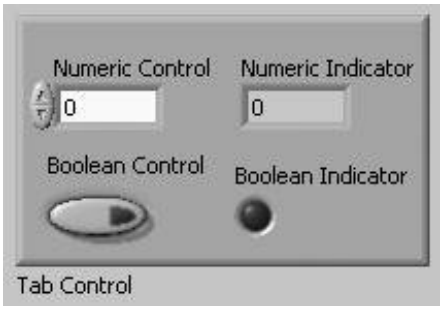
The value of the tab control is the tab that is currently being displayed (frontmost). So, this means that we can programmatically change the visible tab by writing to the value of the tab control (for example, by passing a value to its terminal or a local variable). Note, however, that all controls on all the tabs are always "active"selecting a particular value for the tab control only affects which tab is displayed; it does *not* in any way affect the value of the controls in all the other tabs.

Another feature to note about the tab control is that *it can contain both controls and indicators at the same time*. This is very different from a cluster, which can contain either controls or indicators, but not both.

If you do not want the user to be able to see the tabs of a tab control, you can hide them by unchecking the Visible Items>>Tabs option from the tab control's pop-up menu (or set the Tabs Visible? property to FALSE, programmatically using a property node). Then the tab control will appear similar to a Raised Box decoration. You can programmatically change the value of the tab control, to switch between the pagesthis is a great way to have several "screens" in a single VI.

If you want the tabs of a tab control to be visible, but you do not want the user to be able to switch pages, make the tab control an indicatorthat won't affect the controls or indicators on the pages. (Also, you can't just disable the tabthat disables all the controls on it as well.)

Figure 13.148. Tab control with page tabs hidden



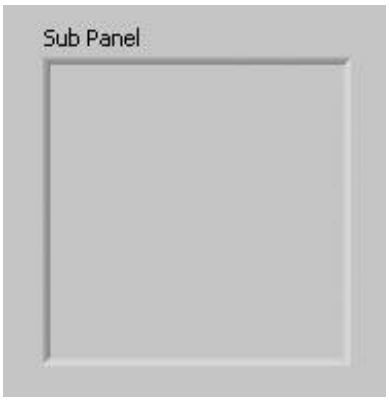
You can create a "wizard" style of user interfaceyou know, the kind with "Back," "Next," and "Finish" buttons that cycle you to different "screens." Hiding the tabs of a tab control and changing the visible page programmatically is a very clever way to create such a user interface. Give it a try! Figure 13.149 shows the front panel of such a VI.

Figure 13.149. A "wizard" user interface created using a tab control with hidden tabs

Subpanels

The subpanel control (found on the Modern>>Containers subpalette of the Controls palette) displays the front panel of another VI, inside a frame that looks similar to a cluster. Figure 13.150 shows an empty subpanel control.

Figure 13.150. Subpanel control on a VI's front panel



On the block diagram, the subpanel has no terminal. Instead, when you drop a subpanel onto the front panel of your VI, LabVIEW creates an invoke node on the block diagram with the Insert VI method selected. This method is used to insert a VI into the subpanel.

To use a subpanel, you will need to provide a VI Reference to the VI you are inserting. VI References are discussed in Chapter 15, where we will learn about the VI Server.

For examples of how to use the subpanel, open some of the VIs that ship with LabVIEW in `examples\general\controls\subpanel.llb`.

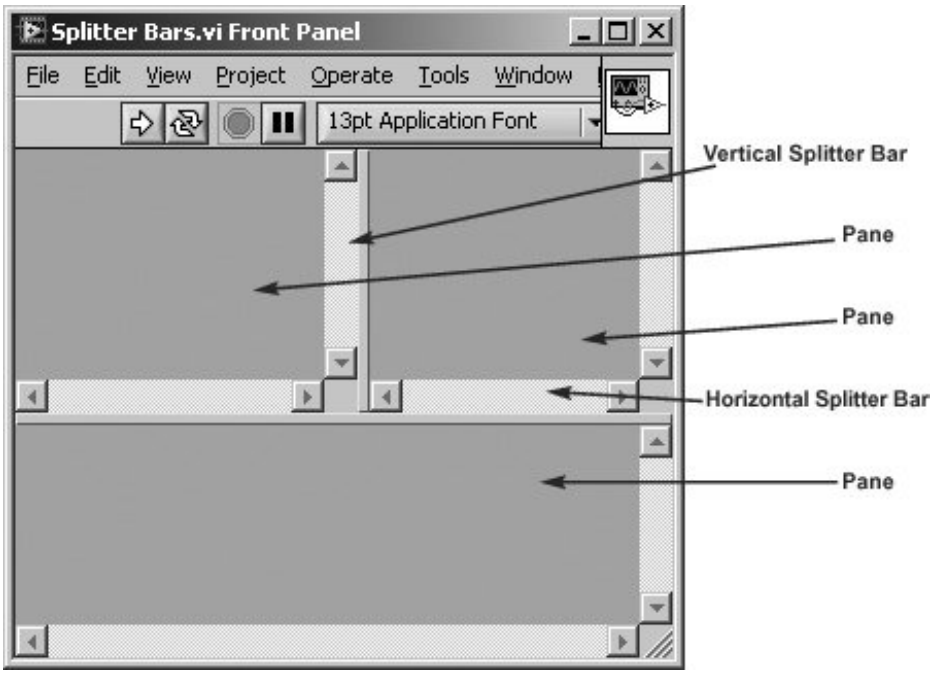
Figure 13.151. Subpanel invoke node on a VI's block diagram

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

Splitter Bars

Splitter bars (found on the Modern>>Containers subpalette of the Controls palette) are used to divide a VI's front panel into *panes*. In fact, a VI's front panel can be thought of as a single pane. Use a *Vertical Splitter Bar* to divide a pane into a *Left Pane* and *Right Pane*. Use a *Horizontal Splitter Bar* to divide a pane into an *Upper Pane* and *Lower Pane*. There is no limit to the number of panes that you can create using splitter bars. Figure 13.152 shows a front panel divided into three panes using one horizontal splitter bar and one vertical splitter bar.

Figure 13.152. A VI Front panel devided into three panes using splitter bars



When you place a control onto the front panel of a VI that is divided into panes, you will place the control into one of the panes. That pane will then own the control.

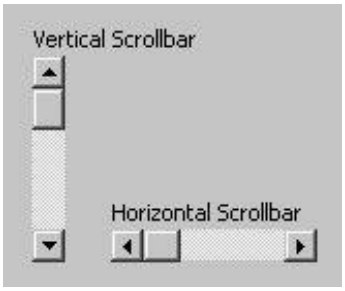
Panes have several properties that affect how the pane size will change as the front panel is size is changed, as well as how the size and position of objects on the pane will change when the pane is resized. You can change a pane's properties from the pop-up menu of an adjacent splitter bar. In the pop-up menu of a vertical splitter bar (not the adjacent scrollbar), there are Left Pane and Right Pane submenus. Likewise, in the pop-up menu of a horizontal splitter bar, there are Upper Pane and Lower Pane submenus.

Splitter bars also have several properties that affect how the user will interact with them and how they behave as the panes are resized. For more examples of how to use splitter bars, open some of the VIs that ship with LabVIEW in `examples\general\controls\splitter.llb`. Also, the LabVIEW documentation has a lot of useful information on this powerful feature.

Scrollbars

Scrollbars can be found on many of LabVIEW's controls. For example, the string control, array control, listbox, tree control, and many others all have built-in scrollbars. But sometimes you only need a scrollbarnot the control that it is built into. For those occasions, use the vertical scrollbar or horizontal scrollbar (found on the Modern>>Numeric subpalette of the Controls palette) shown in Figure 13.153).

Figure 13.153. A vertical scrollbar and horizontal scrollbar



It is important to understand the different parts of a scrollbar. Figure 13.154 shows all of the parts of a scrollbar.

Figure 13.154. Scrollbar controls

One thing that is very important to notice about the vertical scrollbar is that the decrement button is at the top and the increment button is at the bottomthis might seem counter-intuitive, but it will make sense in a second. The vertical scrollbar was originally designed for scrolling through lines (and pages) of text. The value of a scrollbar is a numeric that can be thought of as the line of text at the top of the page that the scrollbar is controlling. If we think about the increment button as incrementing the line number, which is visible at the top of a page full of text, then it all starts to make sense. Conversely, the decrement button decrements the line number that is at the top of the page.

In order to take full advantage of a scrollbar, we will need to set some of its properties using a property node, which can be created from the pop-up menu of the scrollbar (or its block diagram terminal) by selecting a property from the Create>>Property Node submenu.

The following properties will be useful in configuring the behavior of the scrollbar:

- Doc Min The minimum value of the scrolling range. Think of this as the minimum line number.

Usually this should be zero.

- Doc Max The maximum value of the scrolling range. Think of this as the total number of lines in our document.
- Increment The increment value of the scrolling range. When you click the increment and decrement arrows, the scrollbar value adds or subtracts the increment value to move the scroll box toward the arrow that you click.
- Page Size The page size of the scrolling range. When you click the spaces between the scroll box and the arrows, the scrollbar value changes by the page size. Think of this as the number of lines in a page of text.
- Housing Length The length of the housing in pixels. This is the height of a vertical scrollbar and the width of a horizontal scrollbar.

Clicking on the page-down region increments the scrollbar value (or *line number*, if you will) by the page size, and the page-up region decrements the scrollbar by the page size. Also, the size of the scroll thumb (see Figure 13.154), relative to the Housing Length, is equal to the page size (in lines) relative to the total document size (Doc Max Doc Min = Total Number of Pages in Document).

Open `examples\general\controls\Cluster with Scrollbar.vi` for an example of using a vertical scrollbar to simulate a cluster scrollbar.

Graphics and Sound

With LabVIEW being a graphical programming language, it only seems natural that it has the capacity to work with and manipulate front panel graphics.

LabVIEW can works with graphic images on the front panel in two ways:

- Static graphics, such as a logo, can be imported into LabVIEW and placed on the front panel.
- For programmatic generation, control of dynamic graphics, LabVIEW provides the Picture control, a powerful object with a comprehensive set of functions.

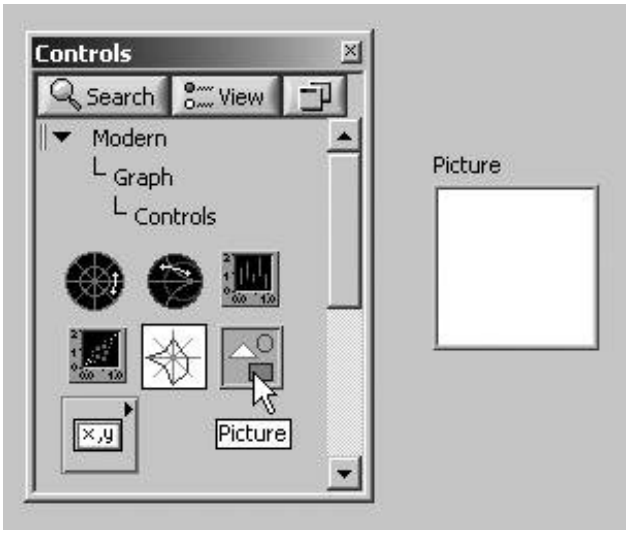
Importing Graphics

LabVIEW allows you to import graphics by copying and pasting from the clipboard (on Windows and Mac OS X only), and using the Edit>>Import Picture from File . . . menu option. You can learn more about importing graphics in Chapter 17.

Picture Control

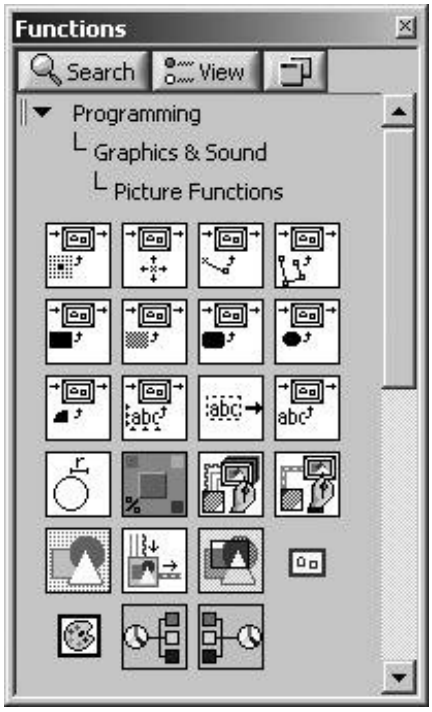
The Picture control (found on the Modern>>Graph>>Controls palette) is a very powerful LabVIEW control for drawing pictures and graphics of anything you can think of (see Figure 13.155).

Figure 13.155. Placing a Picture control onto the front panel from the Modern>>Graph>>Controls palette



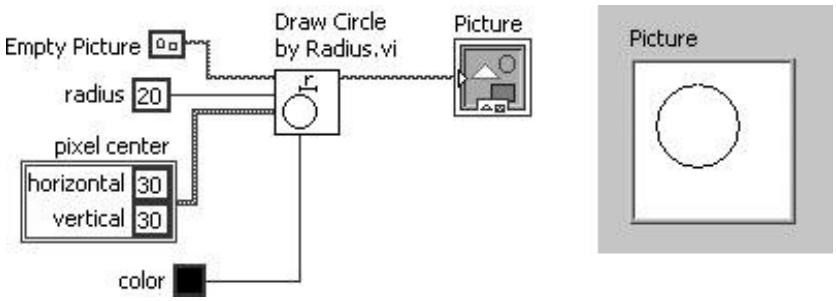
In order to draw pictures in the Picture control, use the picture function located in the Programming>>Graphics & Sound>>Picture Functions palette (Figure 13.156).

Figure 13.156. Picture Functions palette



For example, you can draw geometric shapes (as shown in Figure 13.157), text, lines, images read from filethere's really no limit!

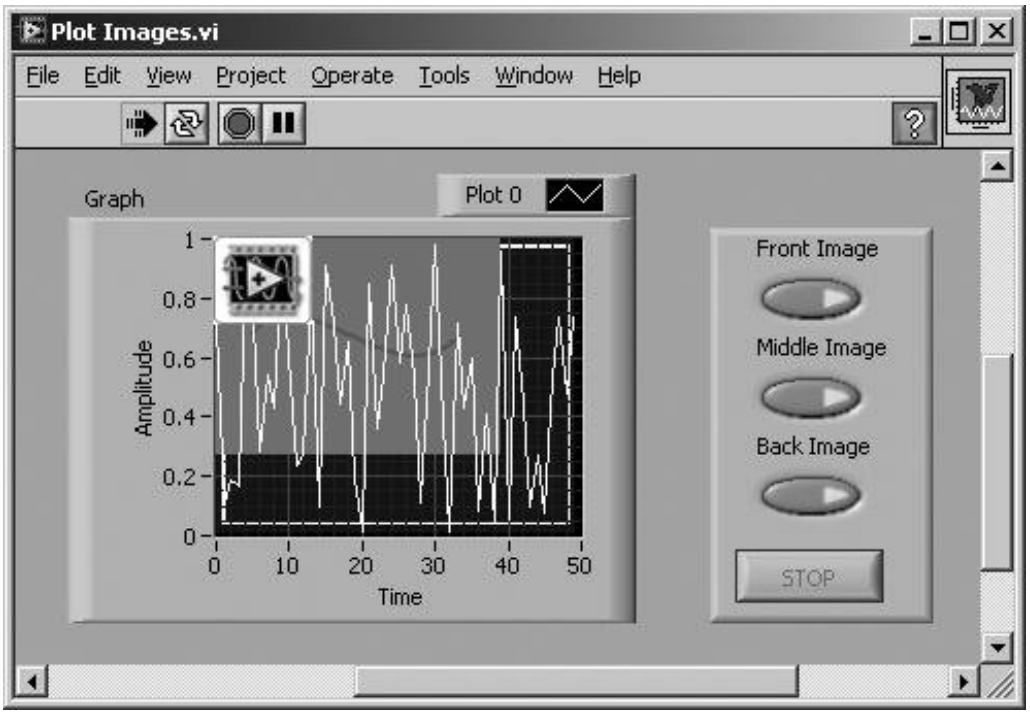
Figure 13.157. Drawing a circle in a picture control



There are a lot of good examples on how to use the picture control in the `examples\picture` folder of your LabVIEW installation.

The picture data type is also very useful for embedding images in the plot area of a graph, as shown in Figure 13.158. This is an example VI that may be found at `examples\general\graphs\Plot Images.vi`.

Figure 13.158. Plot Images.vi front panel



Embedding plot images is easy; just write a picture into one of the following properties of a graph:

- PlotImages.Front is drawn in front of the plot data.
- PlotImages.Middle is drawn between the grid lines and the plot data.
- PlotImages.Back is drawn behind the grid lines.

Sound

A very nice or very obnoxious feature to have in many applications is sound, depending on how you use it. An audible alarm can be very useful in situations where an operator cannot look at the screen continuously during some test. A beep for every warning or notification, however, can be downright annoying. So use sounds with some thought to how often they will go off.

Beep.vi

Beep.vi (Programming>>Graphics & Sound palette) gives you access to the operating system's beep.

LabVIEW also has the ability to record and playback sound files, using the WAVE (.wav) file format. Use the functions found in the Programming>>Graphics & Sound >> Sound subpalette (shown in Figures 13.159 and 13.160) to perform these operations and browse the sound examples with

LabVIEW to get a feel for how they work. Note that the Sound palette and VIs are slightly different in Windows than they are in Linux and Mac OS X.

Figure 13.159. Sound palette (Mac OS X and Linux)

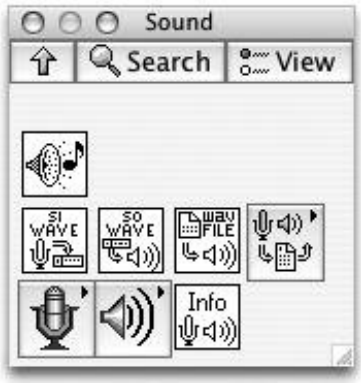


Figure 13.160. Sound palette (Windows)

PREV NEXT

Wrap It Up!

In this chapter, we mined some of LabVIEW's gemstones: local variables, global variables, shared variables, property nodes, and invoke nodes. The power and versatility offered by these structures and functions allow you to go a level deeper into programming applications in LabVIEW.

Local variables allow you to create a block diagram "copy" of a front panel object, which you can read from or write to in different places. Global variables are similar to locals, but they store their data independently of any particular VI, allowing you to share variables between separate VIs without wiring them together. Locals are very useful for controlling parallel loops or updating a front panel object from multiple locations in the diagram. Globals are a powerful structure, but care is required in their use because they are prone to cause problems. Shared variables are useful for storing buffered data that is going to be shared between VIs or across a network.

Property nodes give you immense control over the appearance and behavior of indicators and controls, allowing you to change their properties programmatically. Every control and indicator has a set of base properties (such as color, visibility, etc.). Many objects, such as graphs, have numerous properties that you can set or read accordingly.

The Event Structure is an extremely useful structure that lets you do true event-driven programming, and you'll want to use this often to handle GUI events.

We saw how type definitions are handy for making a "master" copy of a data structure or control you might be using in several places, and how the instances of that typedef can be updated automatically by just updating the typedef control.

You learned that the state machine, using a combination of a While Loop, a Case Structure, and shift registers, offers a foundational programming pattern that virtually all advanced LabVIEW developers rely on.

Messaging and Synchronization functions allow you to send messages across applications. Queues allow one or more message producers to send messages and data to a single consumer. Notifiers allow one or more message producers to send status information and data to one or more message consumers. Semaphores allow shared resources to be locked and unlocked to avoid race conditions and other resource contention issues. Rendezvous allow parallel processes to "meet up" at waypoints before proceeding. Occurrences are low-level dataless event messages that are lightweight and efficient (but with much fewer features than the other messaging functions). The First Call? function tells you when a subVI or block of code is called for the first time, allowing you to conditionally run initialization code only once.

Finally, we examined some cool GUI widgets and learned how to integrate sound and graphics into our application.

14. Advanced LabVIEW Data Concepts

Overview

This chapter will show you how to use some more advanced data structures related to LabVIEW. First we'll briefly look at polymorphic VIs and why they are useful. Then we'll spend some time with advanced file I/O, seeing how to work with both text and binary files. We'll look at a special kind of file common to LabVIEW applications: the configuration file (sometimes called an INI file), and the functions LabVIEW provides for configuration files. In addition, this chapter will show you how LabVIEW can make system calls to the computer, how to import and export external code in the form of DLLs or Shared Libraries, and some more information on converting data types in LabVIEW. We'll introduce the Variant data type. Finally, you'll take a look at advanced data conversions and why you might need them.

Goals

- Understand how a polymorphic VI behaves
- Learn how to use some of the more advanced file I/O VIs
- Discover LabVIEW's functions for configuration (INI) files
- Get an overview of what LabVIEW can do to interface with external code and DLLs in particular
- Know how to perform advanced conversions between different data types
- Become familiar with the Variant data type

Key Terms

- Polymorphic VI
- Polymorphic VI Selector
- File refnum
- File marker
- File position
- Configuration file
- INI
- Key-value pair
- System Exec
- Call Library Function Node
- DLL
- CIN
- Type Cast
- Variant

A Word About Polymorphic VIs

By now you are comfortable with how to use and create subVIs including their connector pane. One concept we want to introduce you to is the *polymorphic VI*. Like the name implies, a polymorphic VI is capable of handling different data types on its input, by automatically adapting to the wired data type. Polymorphic VIs accept different data types for a single input or output terminal. A polymorphic VI is really just a collection of VIs (called members) having the same connector pane pattern. When a polymorphic VI is placed on the block diagram as a subVI, you only "see" one of the member VI icons at any given time.

With LabVIEW's primitive functions, such as the Add function, you are already familiar with how these functions are polymorphic. That is, you can wire scalars, arrays, or clusters to the Add function and it automatically will work with those data types.

Normally, when you create a subVI and wire its connector pane to the inputs and outputs, you are explicitly defining the data types it works with. So, if your subVI takes a numeric value as "input 1," you normally can't wire a string to "input 1" or you'll get a broken wire. But if a VI is created as a Polymorphic VI, you can define multiple types of data for the same subVI inputs and outputs, and you can create separate pieces of code to handle each. For example, the Autocorrelation function (from the Signal Processing>>Signal Operation palette) is a polymorphic VI. When you put it on the block diagram, you can wire either a 1D array or a 2D array to the "X" input (see Figures 14.1 and 14.2). Notice that wiring a 2D array causes one input to disappear and the output to also become a 2D array type.

Figure 14.1. AutoCorrelation VI with 1D array wired

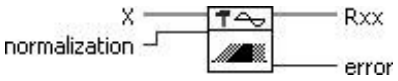
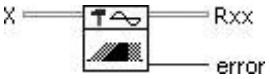
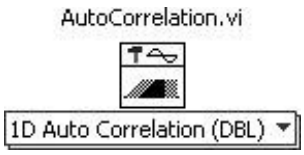


Figure 14.2. AutoCorrelation VI with 2D array input wired



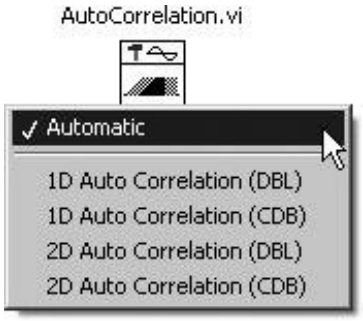
Pop up on a subVI that is polymorphic, and select Visible Items>>Polymorphic VI Selector to show the Polymorphic VI Selector (see Figure 14.3).

Figure 14.3. Polymorphic subVI with Polymorphic VI Selector visible



The Polymorphic VI Selector is a drop-down list showing all of the member VIs that belong to the polymorphic VI (see Figure 14.4).

Figure 14.4. Polymorphic VI Selector drop-down list of members

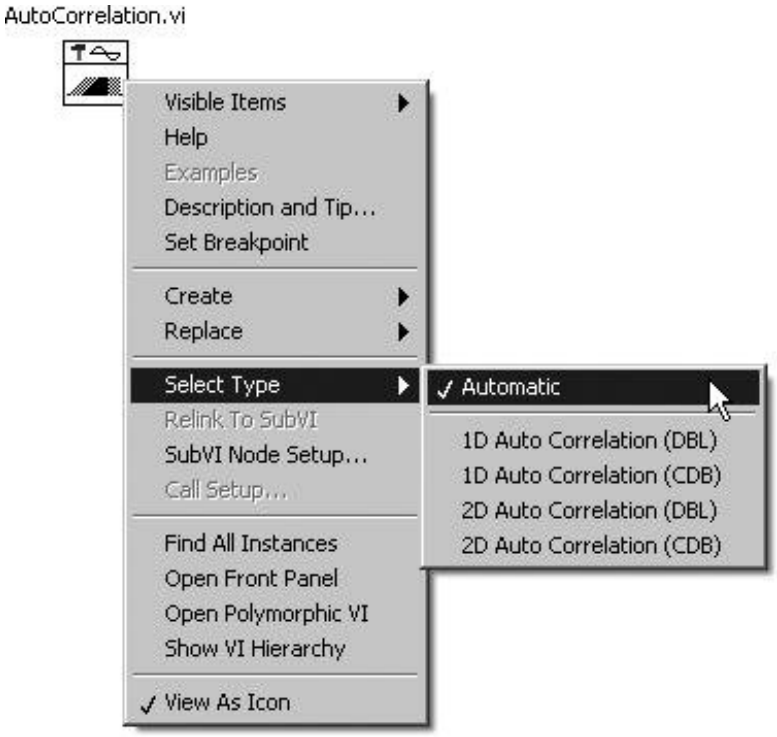


You will notice that the default setting for any polymorphic subVI is *Automatic*. This means that the subVI is configured to automatically adapt to the data type that is wired into it. However, by selecting a different member VI from the Polymorphic VI Selector, you will disable automatic selection and use the specific member that you have selected.

The Polymorphic VI Selector will display the member that is currently selected, whether it was chosen automatically or explicitly. You cannot tell how a polymorphic subVI has been configured simply by looking at it.

There is another way, beside the Polymorphic VI Selector, to configure (and view the configuration of) a polymorphic subVI. Pop up on the subVI and select any of the options beneath the Select Type>> submenu (see Figure 14.5). Note that these are the exact same options available from the Polymorphic VI Selector.

Figure 14.5. Configuring a polymorphic subVI from its pop-up menu



It's beyond the scope of this book to teach you how to make polymorphic VIs, but we wanted you to be aware of their existence, because many of LabVIEW's built-in functions are polymorphic VIs. For example, most of the DAQmx VIs you saw in Chapter 11, "Data Acquisition in LabVIEW," are polymorphic, as they allow you to read or write scalars, arrays, waveforms, and so on, with the same VI function. In this chapter, we'll look at Configuration VIs, polymorphic VIs for reading and writing configuration files.

To learn how to create your own polymorphic VIs, consult the following section of the LabVIEW Help Documentation: Fundamentals>>Creating VIs and SubVIs>>How-To>>Creating SubVIs>>Building Polymorphic VIs.

PREV NEXT

Advanced File I/O: Text Files, Binary Files, and Configuration Files

You've already seen from Chapter 9, "Exploring Strings and File I/O," the most common file I/O functions LabVIEW has to offer. You've seen how to read and write dynamic data types, spreadsheet data, text files, and binary files. You will now learn how to operate on files by reference, performing a sequence of operation and introspection steps, rather than simply reading or writing the file in one fell swoop. This will allow you to achieve some more complex and high-performance use cases.

Opening and Closing File References

A *file* is a set of ordered bytes on your computer disksimilar to the way a string is a set of ordered characters or an array is a set of ordered elements. Because files can be very big, it is often desirable to not read the entire contents of a file into memory all at once, but rather, only read small portions of it (maybe one line at a time) and perform operations based on the data that is read.

Most of the functions for writing and reading data to and from files use a *refnum* (short for "reference number") as a handle to a file that has been opened for writing or reading. By handle, we mean that we are not passing around the data of the file, but rather, something that refers to the open file. You can think of a file refnum like a path to an open fileit is not the data of the file, but something that helps us know where to find the data.

The file read and write functions in LabVIEW are polymorphic. You can use them with either file path inputs OR file refnum inputs. In this section, we'll learn how they work if we are using file refnums.

To open a refnum to a file, we use the Open/Create/Replace File function (see Figure 14.6). The refnum will be passed out via the refnum out terminal. When we are done writing or reading the file, we will pass the file refnum to the Close File function (see Figure 14.7), to tell LabVIEW that we no longer need access to the file.

Figure 14.6. Open/Create/Replace File

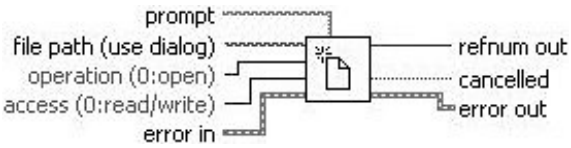
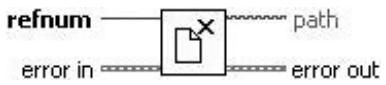


Figure 14.7. Close File



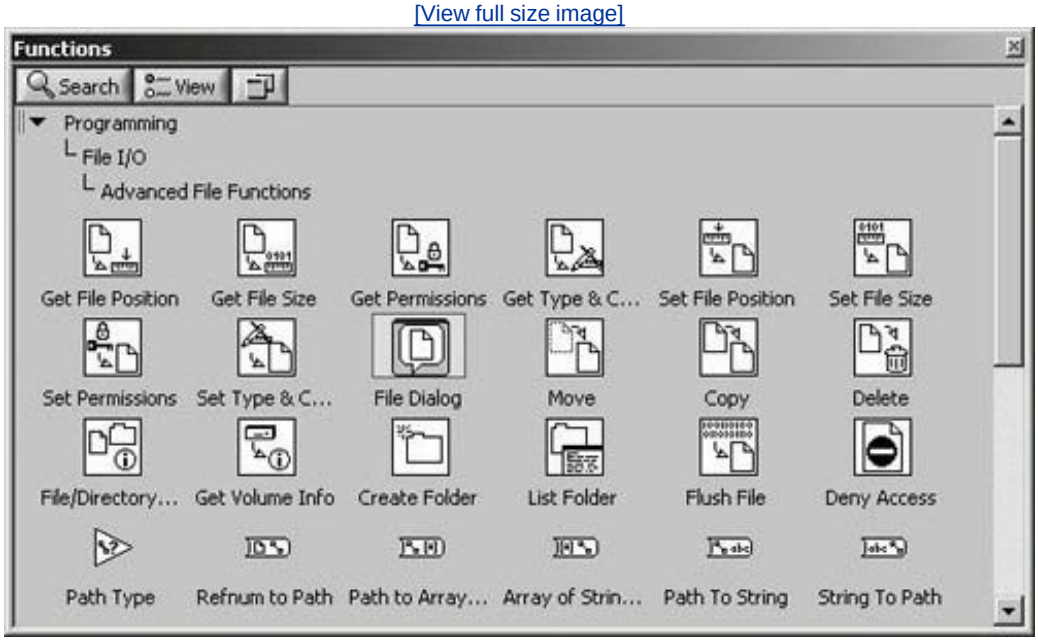
Now that we know how to open and close a file, let's take a look at how to write and read data.

Advanced File Functions

Sometimes you'll need to do specific file tasks, such as create directories, change the permissions on a specific file, or delete a file. LabVIEW gives you many more file I/O functions in the Programming>>File I/O>>Advanced File Functions subpalette, shown in Figure 14.8.

Figure 14.8. Advanced File Functions palette

[View full size image]



We won't cover every one of these functions in this chapter, but we'll discuss a few of them next.

File Position

The *file marker* is a number that tells us the byte offset where you are currently working in a file. The *end of file* is the offset past (just beyond) the last byte in the file (meaning the end of the file)the *end of file* is also the *file size*.

When you first open a file, the file marker is set to an initial value of 0, and as you write or read to the file, the marker is moved forward each time by the number of bytes written or read. The file marker helps you by keeping track of where you are in the filethere is no need for you to explicitly specify the position where you want to write or read data (although you *can* specify the position to read or write, if you wish). Generally, you will start at the beginning of a file and keep writing or reading data contiguously until you reach the end of the filethis is typically referred to as *sequential access*, as compared to *random access* (explicitly specifying the position of each read and write operation).

For example, if you open a file, the *file marker* is initialized to 0. Now, if you write 100 characters to the file, the write operation will start at byte 0, and after the write operation, the new value of the file marker will be 100. Then, if you write 50 more characters to the file, the write operation will start at 100 (the current file marker value) and when you are done, the new file marker value will be 150. Pretty easy, isn't it?

So, LabVIEW automatically adjusts the file marker as you write or read the file, but if you wish, you can also move the file marker explicitly to any value you like (so that read and write operations will occur in this new location), using the Set File Position function, shown in Figure 14.9. Use the offset input to define the marker position, and the from input to specify whether the offset value is relative to the *start of the file*, *end of the file*, or the *current file marker position*. Use the Get File Position function, shown in Figure 14.10, to get the current value of the file marker.

Figure 14.9. Set File Position

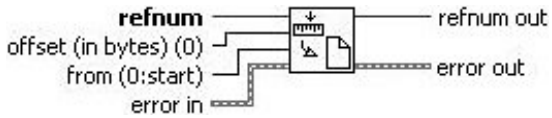
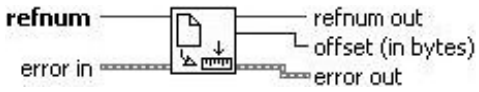


Figure 14.10. Get File Position



You can use data files on disk, in a similar way to how you might use data arrays in memorythis is a useful fact because data files can be very large compared to the amount of data you can keep in memory (RAM). For example, when you build an array, you append elementswhen you build a file, you append data. You can replace array elements by specifying an index and writing the new elementsyou can replace data in a file by specifying the offset and writing the data (which overwrites the existing data). You can read array elements by specifying their indexyou can read data from a file by specifying the offset to the data. So, if you need to operate on HUGE data sets, consider using binary files without loading the whole thing into memoryjust read and write the portions you need at a given time.

Normally you don't need to manually move the file marker around, but it's good to know you have that option if you are doing some specialized or low-level file I/O.

End of File

The *end of file* is the offset past the last byte in the filenamely, the *file size*. Use the Set File Size and Get File Size functions, shown in Figure 14.11 and Figure 14.12, to set and get the *end of file* (respectively).

Figure 14.11. Set File Size

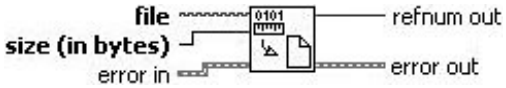
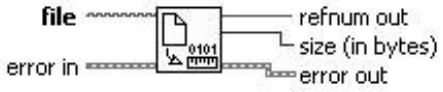


Figure 14.12. Get File Size



As you write data to a file, the end of file is automatically adjusted if the write operation would move the file marker past the end of file (meaning that the file will grow in size). Increasing the file size using the Set File Size function will cause the file to grow, and be padded with null data (zeros). Conversely, if the file size is reduced by setting the end of file to a value smaller than its current value, the data between the new end of file and the original end of file will be clipped (deleted).

Moving, Copying, and Deleting Files and Folders

The following functions allow you to move, copy, and delete both files and folders. Move will move a file from the source path to the target path (see Figure 14.13).

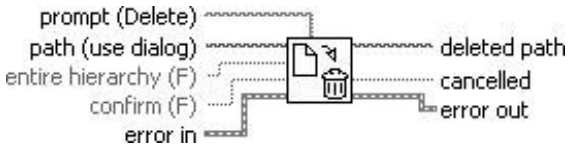
Figure 14.13. Move

Copy copies a file from the source path to the target path (see Figure 14.14).

Figure 14.14. Copy

Delete deletes a file or directory (see Figure 14.15). WARNING!!! Be very careful using this function. It can delete files or whole directories without any prompting!!! That reminds us, when was the last time we backed up our data?

Figure 14.15. Delete

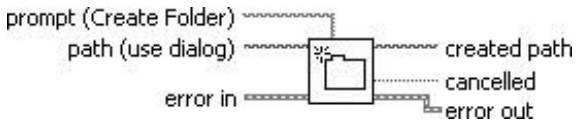


Creating Folders and Listing Folder Contents

The following functions allow you to create folders and list the files and subfolders contained in folders.

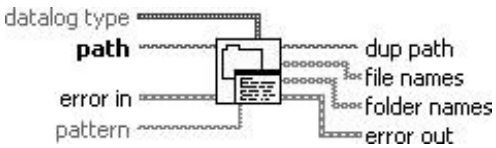
Create Folder will create a directory specified by the path (see Figure 14.16). Note that this function will create multiple parent directories, if they do not already exist. For example, if "C:\Data" does not exist and you attempt to create the directory "C:\Data\Run01," this function will first create "C:\Data" and then "C:\Data\Run01."

Figure 14.16. Create Folder



List Folder returns the contents of a directory specified by path (see Figure 14.17). You can also optionally specify a file pattern to filter the results.

Figure 14.17. List Folder



Activity 14-1: Reading a Text File

You will create a VI that reads the entire contents of a text file. It will display the file's contents in a string indicator and the file's size in a numeric indicator.

1. Build the front panel shown in Figure 14.18.

Figure 14.18. Front panel of the VI you will create during this activity

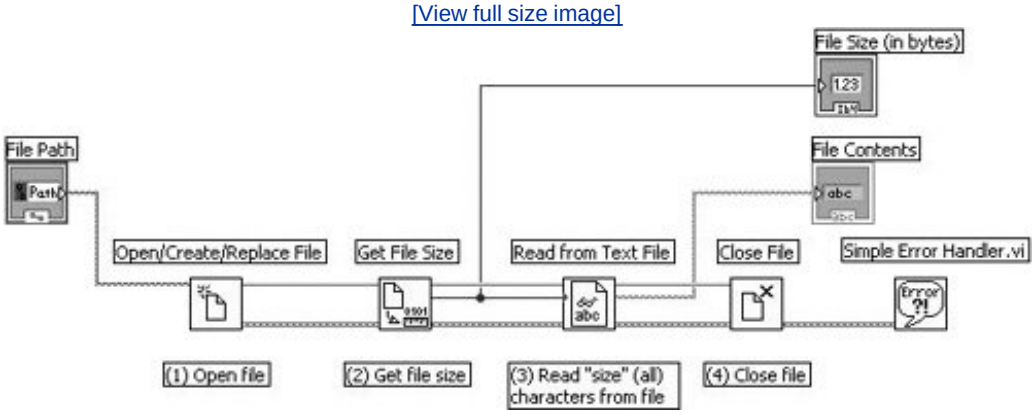
[View full size image]

The VI will read the contents of the file specified by the File Path control and display the output in the string indicator. The digital indicator will display the file's length.

2. Build the block diagram shown in Figure 14.19.

Figure 14.19. Block diagram of the VI you will create during this activity

[View full size image]



Open/Create/Replace File Function

Open/Create/Replace File function (Programming>>File I/O palette) opens the file and returns a refnum that will be used by the other File I/O VIs.

Get File Size Function

Get File Size function (Programming>>File I/O>>Advanced File Functions palette) returns the size of the file, which will be used for specifying the number of characters to read.

Read from Text File Function

Read from Text File function (Programming>>File I/O palette) returns the number of characters from a file specified by the count input, starting at the current file position (which is zero, in our case). Note that this function is polymorphic. You can wire either a file refnum as we are doing here, OR a file path as we did in Chapter 9, "Exploring Strings and File I/O." If you wire a path and do not wire refnum out, the file reference will be automatically closed.

Close File Function

Close File function (Programming>>File I/O palette) closes the file refnum that was created by Open/Create/Replace File.

Simple Error Handler.vi

Simple Error Handler.vi (Programming>>Dialog & User Interface palette) displays a dialog if there is an error in one of the file I/O functions.

3. Return to the front panel and enter a path to a text file into the file path control. Make sure that you don't select a file that is too large (which will cause LabVIEW to use a lot of memory), or is a binary file (which will just display funny characters).

4. Run the VI and look at the file contents that are displayed in the string indicator, and the file size that is displayed in the numeric indicator.

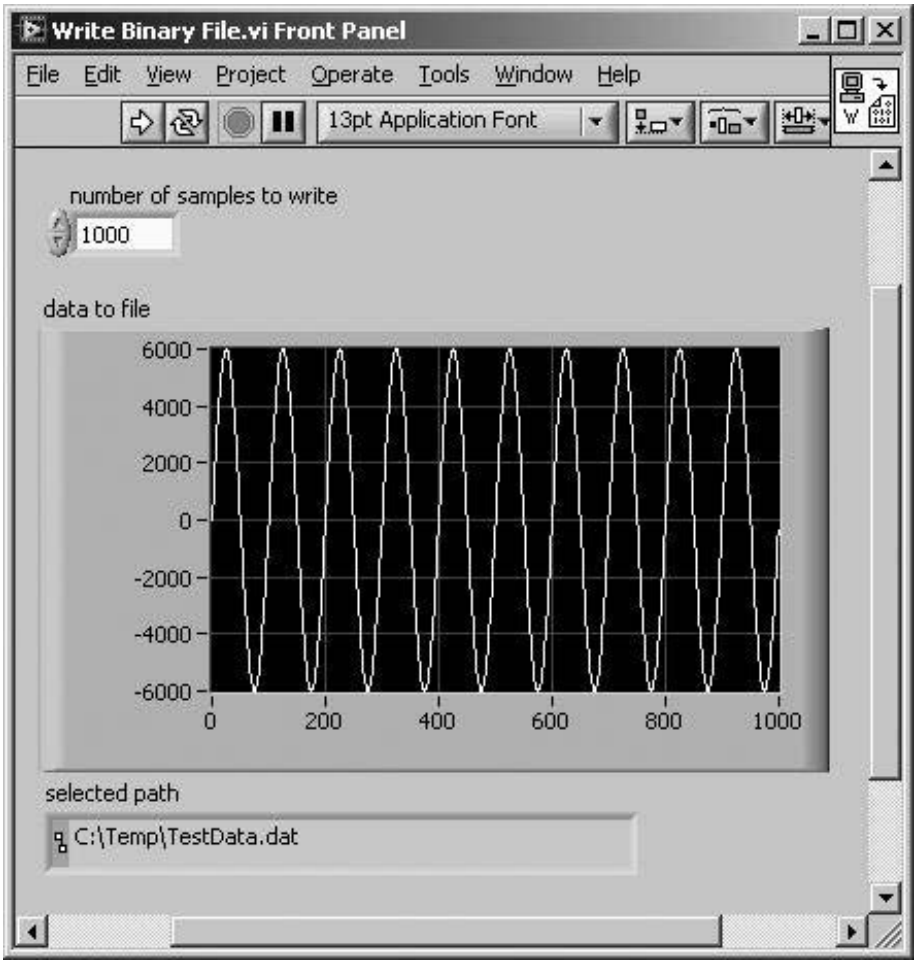
5. Save and close the VI. Name it `Read Text File Advanced.vi` and place it in your `MYWORK` directory. Great jobyou're just made a text file viewer!

Activity 14-2: Writing and Reading Binary Files

In this activity, you'll use the file I/O functions to write a binary file, and then read back that file at arbitrary points in the file.

1. First, create a VI called `Write Binary File Advanced.vi` with a front panel like the one shown in Figure 14.20.

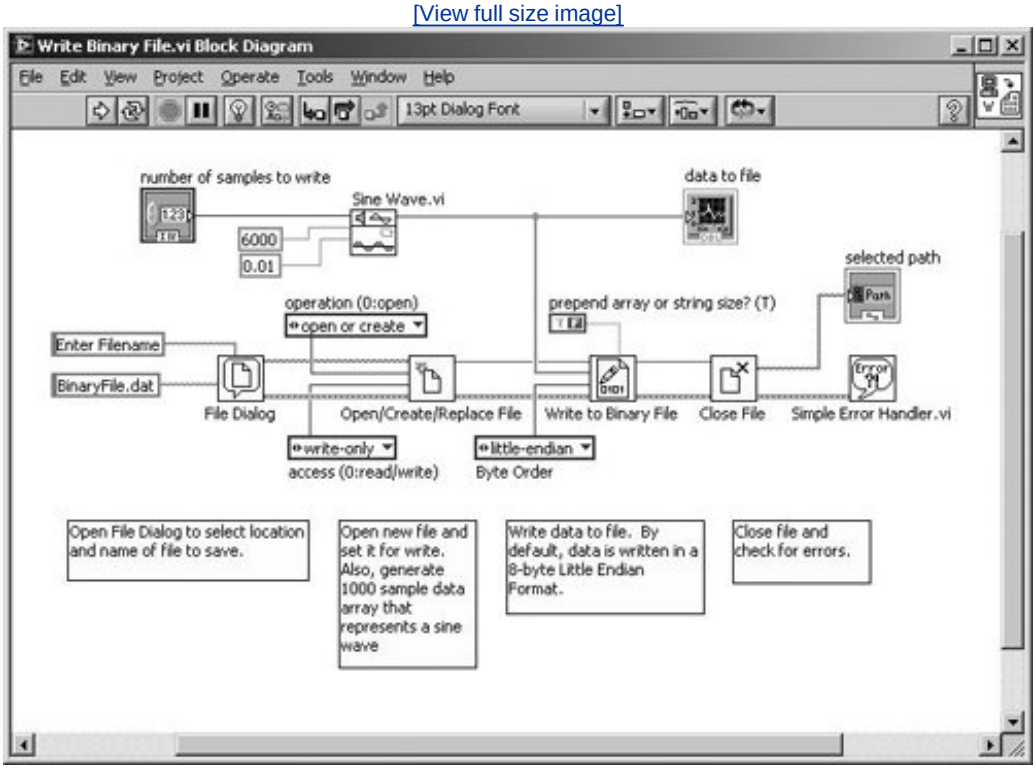
Figure 14.20. Write Binary File Advanced.vi front panel



2. Create the block diagram shown in Figure 14.21. You can use the Sine Wave.vi (found in the Signal Processing >>Signal Generation palette) to generate an array that represents a sine wave pattern. Use the File I/O VIs from the Advanced palette to handle the file dialog, opening and creating the file, and closing the file.

Figure 14.21. Write Binary File Advanced.vi block diagram

[View full size image]



Notice that we write an array of DBL (double-precision floating point numbers) to the binary file. It's important that we know this, and that we know that each DBL point takes up 8 bytes on disk, when it comes time to read back this file.

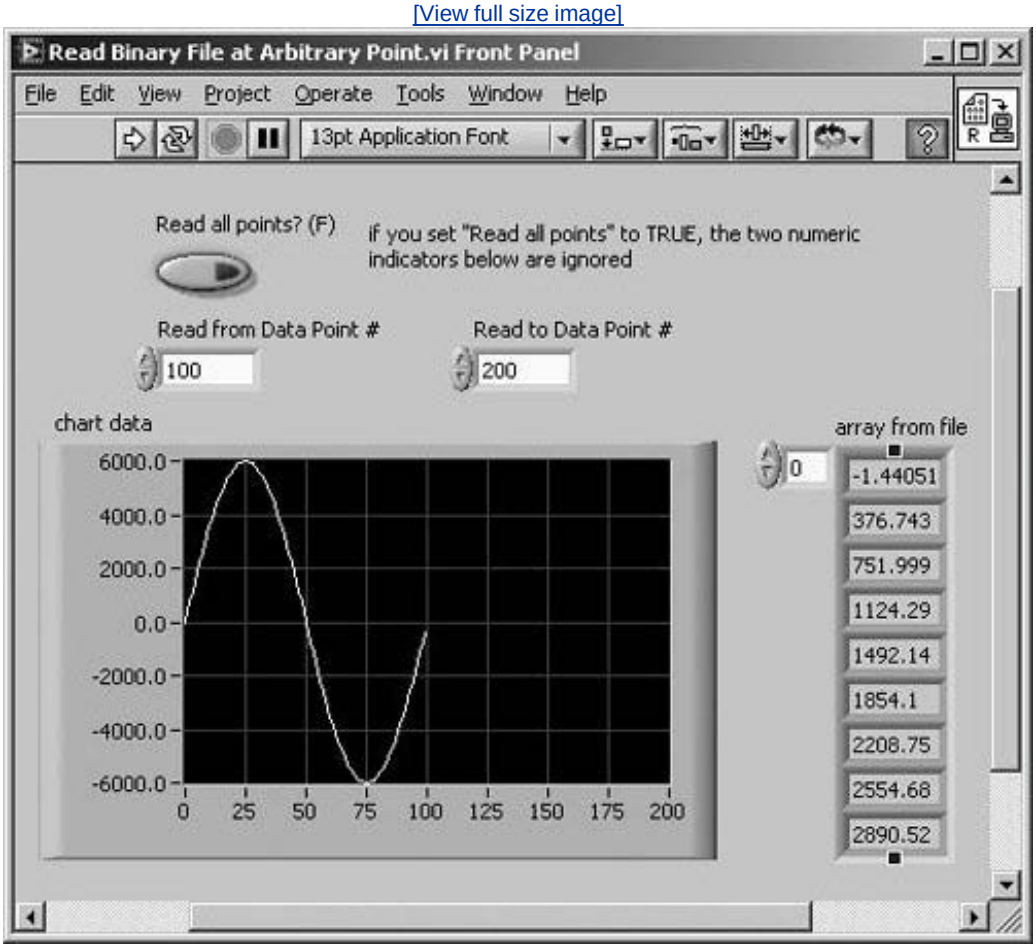
You can find out how many bytes each data type in LabVIEW uses from the Fundamentals>>How LabVIEW Stores Data in Memory section of the LabVIEW Help. (Open the LabVIEW Help by selecting Help>>Search the LabVIEW Help from the menu.)

Run the Write Binary File Advanced.vi and verify it works. Save your binary file somewhere easily accessible.

3. Now let's create the VI to read it. Suppose that you wanted a VI that didn't necessarily read the entire file, but gave you the option of reading from an arbitrary point to any other arbitrary point. For example, if your file has stored 1,000 points (not bytes) of data, you might want to read points 100 through 200. Create the front panel shown in Figure 14.22 and save the VI as `Read Binary File at Arbitrary Point.vi`.

Figure 14.22. Read Binary File at Arbitrary Point.vi front panel

[View full size image]



4. The key to being able to read at arbitrary points in the binary file is to use the Set File Position function (refer to Figure 14.9). You can use this function to set the file mark position. Keep in mind the file mark position must be figured in bytes, not "data points." So in our case, because our binary file is an array of DBL numbers, we have 8 bytes for each data point. If you want to set the file mark to data point position 100, you need to set the file mark at 800 bytes.


Set File Position

5. To read the remaining number of data points, we need to tell Read from Binary File how many points to read. There's an input called count on this function that behaves in two different ways:

   - If you wire the "data type" input (recommended), then the "count" input assumes you're telling it to count data points (elements), *not bytes*. It will automatically figure out how

many bytes because it knows the data type.

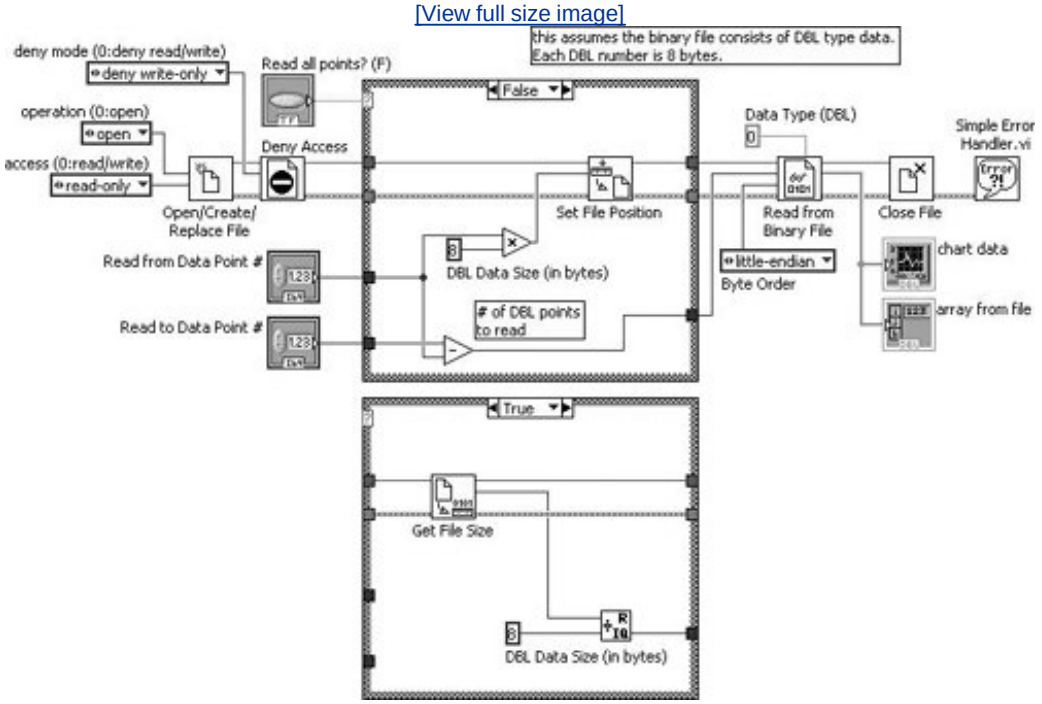
Read from Binary File

- If you do not wire the "data type" input, then the "count" input assumes you're telling it how many bytes to read, because it doesn't "know" what a data point would be in this file anyway.

6. Can you do this activity without peeking at the block diagram shown in Figure 14.23? (Note that the TRUE case of the Case Structure is shown for illustration purposes.) Give it a try; you can always come back here to check your work.

Figure 14.23. Read Binary File at Arbitrary Point.vi block diagram

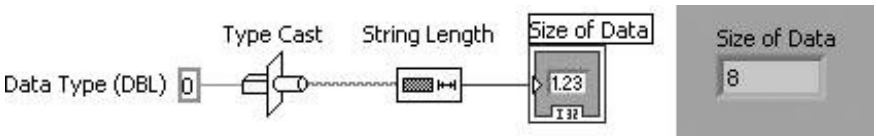
[View full size image]



If you come back to this assignment after reading about the Type Cast function later in

this chapter, you may want to change this diagram to programmatically calculate the size of the data element (a DBL, in this case) instead of using a numeric constant in multiple locations. You can use the String Length function to determine the number of bytes in the flattened data (output by Type Cast) of the data element type, as shown in Figure 14.24.

Figure 14.24. Type casting a DBL numeric to a string and checking the length (number of bytes)



This might not seem to be as easy as just wiring a numeric constant to each location that requires the data length value, but if we ever change the data type of our binary file (for example, from a DBL to a EXT), then we will have to update each of those constants. We might forget about this, and so each of those constants is a potential bug, and thus, a liability.

It might seem that this trick would make the code slower (due to the extra processing steps), but LabVIEW's compiler can determine whether a deterministic calculation is being performed from a constant and replace the calculation with the derived constant (in the compiled code)in this case, an I32 constant equal to 8. This feature is referred to as "constant folding" and is common to most modern compilers.

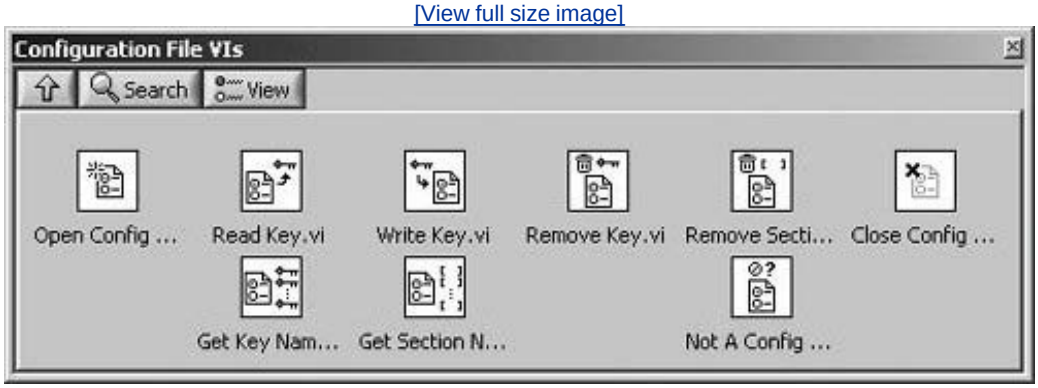
PREV NEXT

Configuration (INI) Files

The Configuration File VIs (Programming>>File I/O>>Configuration File VIs palette) shown in Figure 14.25 allow you to read and write data as key-value pairs to a special file format generally referred to as a configuration or initialization file. Configuration files usually have the file extension `.ini` (short for "initialization"), and are often simply referred to as "INI" or "config" files.

Figure 14.25. Configuration File VIs palette

[View full size image]



Configuration, or INI, files are commonly used in software applications to store important variables that are specific to a software installation.

A typical INI file might look something like the following:

```
[section1]

HardwareType = USB
MonitorResolution = 1024x768
key3 = value3
key4 = value4

[section2]

key1 = value1
key2 = value2
```

As you can see, the file contents are human readable and easy to decipher.

The elements of an INI file are as follows:

- Sections Section start with '[' and end with ']' as in [section1] and [section2] in the preceding.
- Key-value pairs The "HardwareType = USB" is an example of a key-value pair, which begin with a key ('HardwareType'), followed by an equals sign ('='), and a value ('USB').

There is no official file format for configuration files. For example, sometimes people use a semicolon (;) as the first character of a line that contains a comment; however, the LabVIEW Configuration File VIs do not support this feature.

Any configuration file created using the LabVIEW configuration file VIs can be read from or written to on any platform (Mac OS X, Windows, or Linux). So, this is an excellent choice for storing your application's configuration parameters. Additionally, if your application stores configuration parameters in a file, rather than constants on the block diagram, you can change your application's configuration without editing the code. That's a HUGE time saver!

Opening and Closing Configuration Files

You can open a reference to a configuration file, using Open Config Data.vi (Programming>>File I/O>>Configuration File VIs palette), shown in Figure 14.26. If you are creating a new file, make sure to wire a Boolean constant TRUE to the create file if necessary? input. When you are done accessing the configuration file, use Close Config Data.vi (Programming>>File I/O>>Configuration File VIs palette), shown in Figure 14.27, to close the configuration file reference.

Figure 14.26. Open Config Data.vi

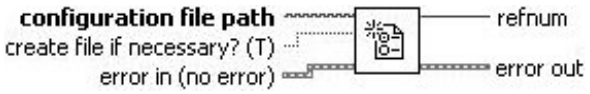
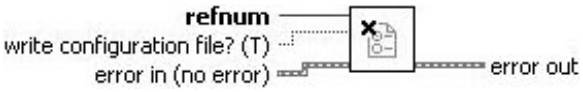


Figure 14.27. Close Config Data.vi



Any changes you make to a configuration file, using the Configuration File I/O VIs, are only made in memory (not saved to disk) until you save the configuration file by calling Close Config Data.vi with the write configuration file? input set to TRUE.

Writing and Reading Key Values

To write and read values to and from keys of a configuration file, use Write Key.vi and Read Key.vi (Programming>>File I/O>>Configuration File VIs palette), shown in Figures 14.28 and 14.29.

Figure 14.28. Write Key.vi

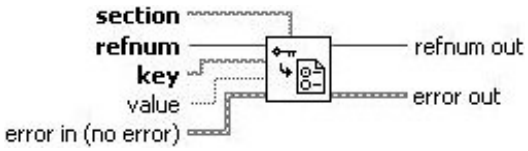
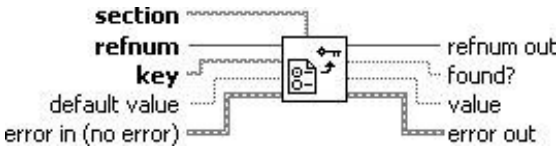
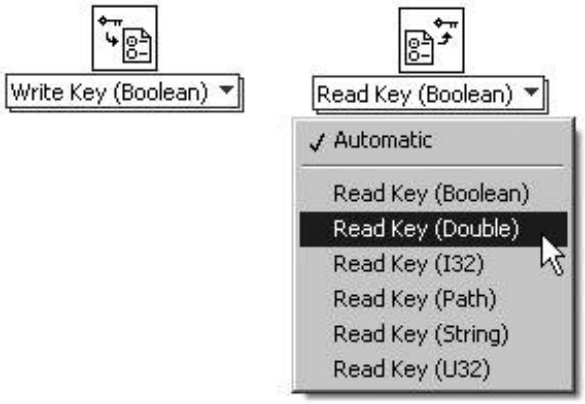


Figure 14.29. Read Key.vi



Both of these VIs have section and key inputs, which are used to specify the key you wish to write or read. These two VIs are polymorphic VIs (which we learned about at the beginning of this chapter) and support values of Boolean, DBL, I32, Path, String, and U32 types. To change the type, simply connect a wire of the desired type to the value input of Write Key.vi, or the default value input of Read Key.vi and the VIs will automatically adapt to the wire type. You can also explicitly choose the type using the Polymorphic VI Selector, as shown in Figure 14.30. You can make the Polymorphic VI Selector visible, by right-clicking on the VI and choosing Visible Items>> Polymorphic VI Selector.

Figure 14.30. Using the Polymorphic VI Selector to explicitly choose a type of a polymorphic subVI



If you want to write a cluster or array to a configuration file, it can take a lot of work to unbundle and index these compound and complex data structures. However, OpenG (http://openg.org) has developed a library of Open Source VIs that can write and read anythingyes, anythingto and from a configuration file! This library is called the Variant Configuration File I/O VIs. See Appendix C, "Open Source Tools for LabVIEW: OpenG," for more information about OpenG and how to obtain this library of VIs.

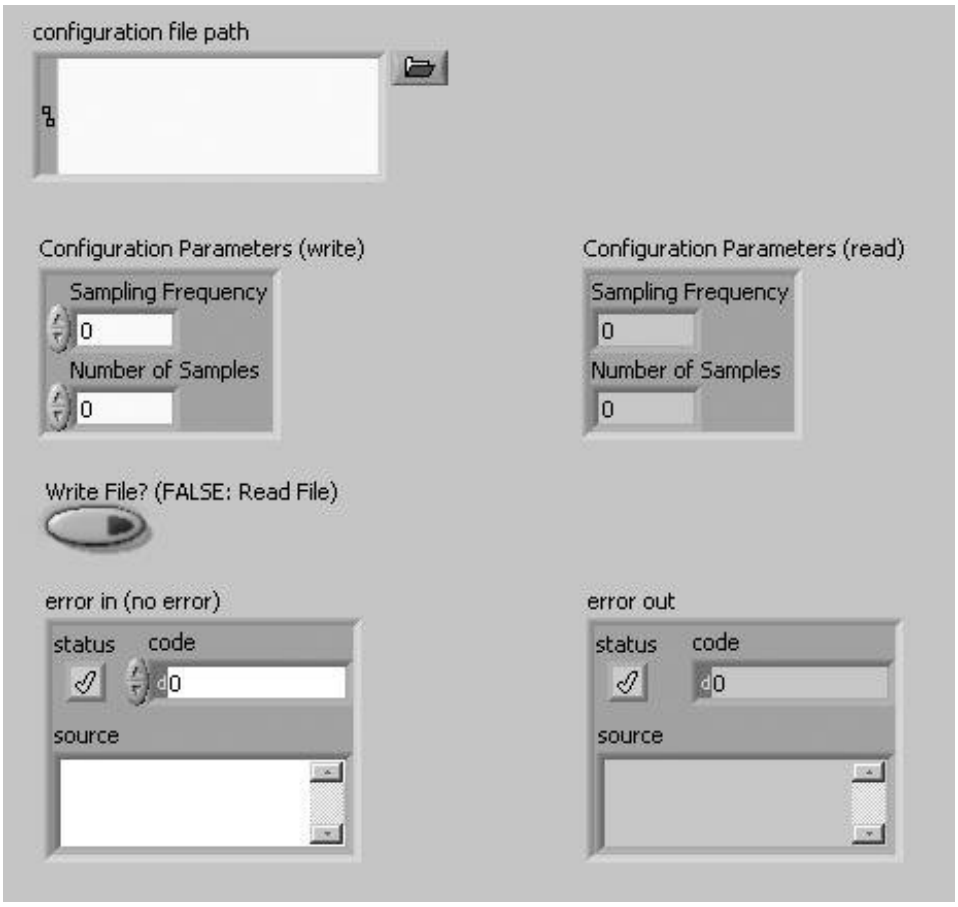
Storing large data sets in configuration files (for example, by flattening a large cluster or array to string and writing the data using Write Key String.vi) is not recommendedthis is not what they are intended for. For large data sets, you will notice (sometimes significant) limitations in performance. If you need to store large data sets in a file, consider using a binary or datalog file.

Activity 14-3: Storing Data in a Configuration File

You will write a very powerful VI that can write and read data to and from a configuration file. This activity might take a little bit of time to complete, but this VI is so useful that it will certainly become a cornerstone of your LabVIEW toolbox.

1. Open a new VI and create the front panel shown in Figure 14.31. Note that inside the `Configuration Parameters` clusters, `Sampling Frequency` is a DBL and `Number of Samples` is an I32.

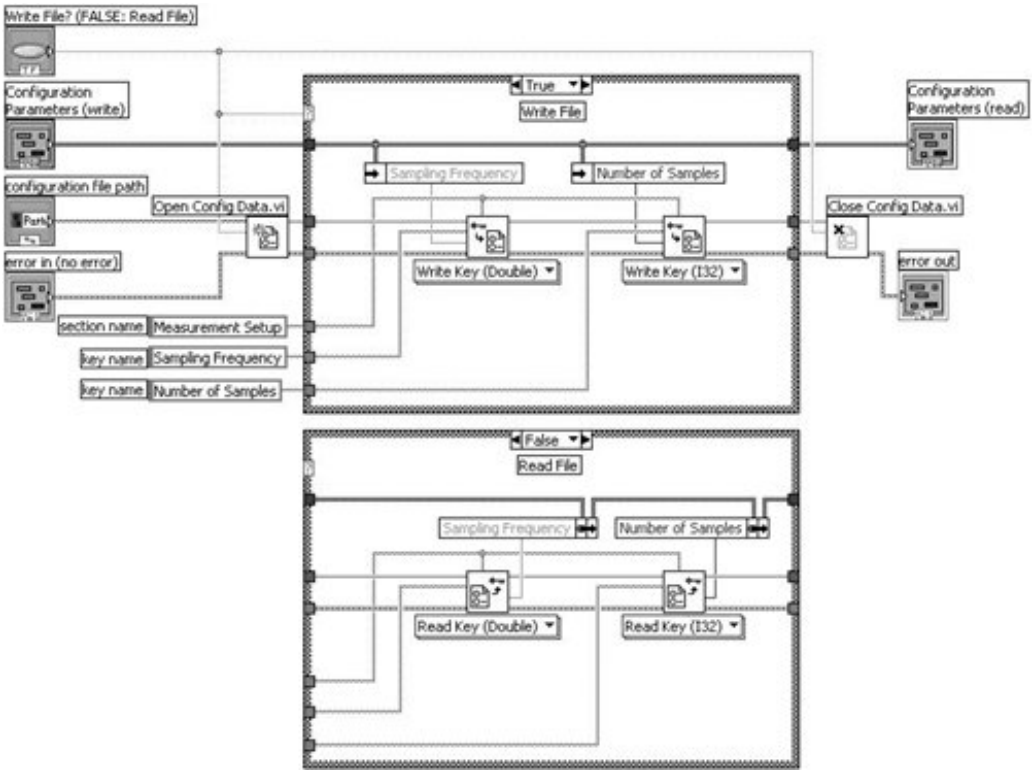
Figure 14.31. Front panel of the VI you will create during this activity



2. Create the block diagram shown in Figure 14.32. Note that the "False" frame of the Case Structure is shown below the "True" frame, only for illustrative purposes.

Figure 14.32. Block diagram of the VI you will create during this activity

[View full size image]



Open Config Data VI

The Open Config Data VI (Programming>>File I/O>>Configuration File I/O palette) opens a reference to the configuration file. Note that the `Write File?` control terminal is wired to the `create file if necessary?` input terminal of Open Config Data. This will ensure that if you are writing the file, it will be created if it does not already exist. However, if you are reading the file, you do not want to create it if it does not exist; rather, you want an error to be generated if the file does not exist.

Write Key VI

The Write Key VI (Programming>>File I/O>>Configuration File I/O palette) is used to write the `Sampling Frequency` and `Number of Samples` keys to the configuration file. The Polymorphic VI Selector is made visible, by right-clicking on the VI and choosing Visible Items>>Polymorphic VI Selector. The instance of Write Key that writes the `Sampling Frequency` has been set to Write Key (Double), and the instance that writes the `Number of Samples` has been set to Write Key (I32).

Unbundle By Name Function

The Unbundle By Name function (Programming>>Cluster & Variant palette) is used to unbundle the elements from the `Configuration Parameters` cluster, which are to be written to the

configuration file.

Read Key VI

The Read Key VI (Programming>>File I/O>>Configuration File I/O palette) is used to read the `Sampling Frequency` and `Number of Samples` keys from the configuration file. The Polymorphic VI Selector is made visible, by right-clicking on the VI and choosing Visible Items>>Polymorphic VI Selector. The instance of Read Key that reads the `Sampling Frequency` has been set to Read Key (Double), and the instance that reads the `Number of Samples` has been set to Read Key (I32).

Bundle By Name Function

The Bundle By Name function (Programming>>Cluster & Variant palette) is used to bundle the key values, read from the configuration file, into the `Configuration Parameters` cluster.

Close Config Data VI

The Close Config Data VI (Programming>>File I/O>>Configuration File I/O palette) closes the reference to the configuration file, and optionally saves the configuration data in memory to disk. Note that the `Write File?` control terminal is wired to the `create configuration file?` input terminal of Close Config Data. This will cause the file to be written, only if `Write File?` is set to TRUE. If you are only reading the file, you do not want to resave it.

3. Enter a path into the `configuration file path` control, set the `Write File?` Boolean to TRUE, and run your VI.

4. Open the file you just created in your favorite text editor application. The file contents should look something like the following:

   ```
   [Measurement Configuration]
   Sampling Frequency=50000.000000
   Number of Samples=1000
   ```

   Edit the values of either of the two keys (for example, change "Number of Samples" from "1000" to "2000") and then save the file.

5. Now, set the `Write File?` Boolean back to FALSE, and run your VI. The new values in the file will appear in the `Configuration Parameters (read)` cluster.

6. Congratulations! You just built a power tool that you can reuse in all of your applications. You can customize it, by adding additional keys into the `Configuration Parameters` clusters.

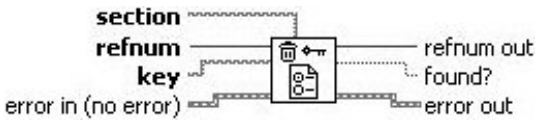
7. Save the VI in your MYWORK directory as Read-Write Configuration File.vi.

Additional Configuration File Operations

If you are doing more than just writing and reading keys to and from a configuration file, the following VIs will be of great use to you. They allow you to remove sections and keys, as well as list keys and sections present in the configuration file.

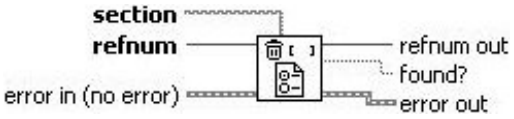
Remove Key.vi (Programming>>File I/O>> Configuration File VIs palette) deletes the specified key, from the specified section of the configuration file (see Figure 14.33).

Figure 14.33. Remove Key.vi



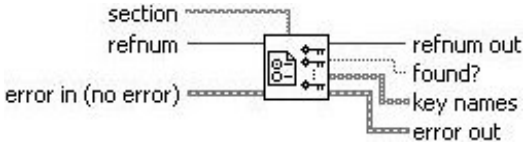
Remove Section.vi (Programming>>File I/O>>Configuration File VIs palette) deletes the specified section, along with all of its keys, from the configuration file (see Figure 14.34).

Figure 14.34. Remove Section.vi



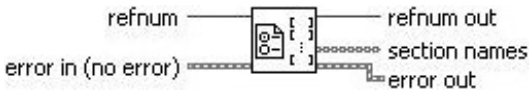
Get Key Names.vi (Programming>>File I/O>>Configuration File VIs palette) returns the names of all the keys in a specified section of the configuration file (see Figure 14.35).

Figure 14.35. Get Key Names.vi



Get Section Names.vi (Programming>>File I/O>>Configuration File VIs palette) returns the names of all the sections in the configuration file (see Figure 14.36).

Figure 14.36. Get Section Names.vi



PREV NEXT

Calling Code from Other Languages

What happens if you *have* to use some code written in another language (such as C, C11, or Visual Basic)? Or if, for some reason, you just miss typing in all those semicolons in your familiar text-based code? LabVIEW does give you some options for interfacing with code from other languages. If you are thinking about writing *all* your code in C or C11, you should check out LabWindows/CVI (available from National Instruments), a programming environment very similar to LabVIEW; the main difference is that C code replaces the graphical block diagram. But if you'd like to (dare we say it?) actually have *fun* programming, then stick to LabVIEW and use conventional code only when you have to.

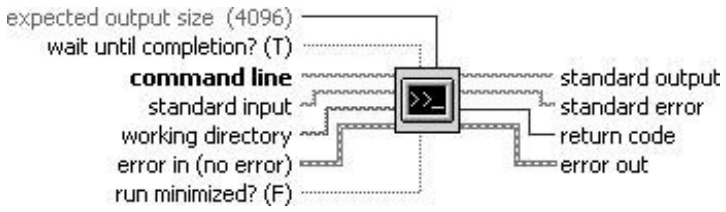
You may want to skip this section if you aren't familiar with writing code in other programming languages such as C, or if you don't expect to need to interface LabVIEW and external code.

LabVIEW gives you three options to call external, compiled code:

- Using the command line with System Exec.vi.
- Calling dynamic linked libraries, or DLLs (DLLs apply only to Windows; for Mac OS X and Linux, you can call Frameworks and Shared Libraries, respectively) with the Call Library Function Node.
- Interfacing to compiled code using a Code Interface Node.

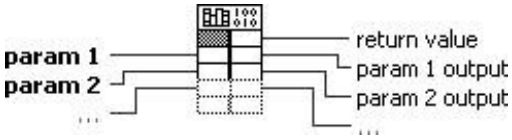
Sysetm Exec.vi is the simplest option to launch a separate executable or program that does what you need. On Windows, Linux, and Mac OS X systems, you do this with the System Exec function (Connectivity>>Libraries & Executables palette), shown in Figure 14.37.

Figure 14.37. System Exec.vi



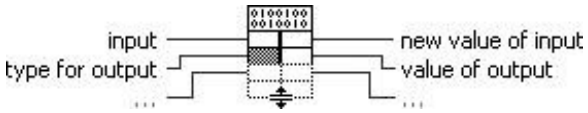
If you have dynamically linked libraries: Dynamic Link Libraries (`.dll`) on Windows, Shared Libraries or Shared Objects (`.so`) on Linux, and Frameworks (`.framework`) on Mac OS X, you can use the Call Library Function Node in LabVIEW (Connectivity>>Libraries & Executables palette), shown in Figure 14.38.

Figure 14.38. Call Library Function Node



Finally, if you want to write your own C code and embed it into LabVIEW, you should use the Code Interface Node (CIN) (Connectivity>>Libraries & Executables palette), shown in Figure 14.39.

Figure 14.39. Code Interface Node



This section talks about interfacing with lower-level compiled code in other languages, but you can also use LabVIEW to communicate with inter-application frameworks like ActiveX and .NET. You'll learn about these in Chapter 16, "Connectivity in LabVIEW."

LabVIEW can also go the other direction. If you need your external code to call a LabVIEW program, LabVIEW can export a VI into an executable (.exe), dynamic link library (DLL) on Windows, shared library (.so) on Linux, and framework (.framework) on Mac OS X. We discussed this capability briefly in Chapter 3, "The LabVIEW Environment."

Using the Call Library Function Node to Call DLLs

We'll look into a little more detail at one of the more common ways you might need to call external code: when it is provided in the form of a DLL.

DLLs are small, compiled libraries in a .dll file that encapsulate some functionality and export a set of *functions* that other applications (like LabVIEW) can call. Usually, you need good documentation to be provided with a DLL for you to be able to use it, because you need to know what the functions are and how the data should be passed to and from them.

Although we will refer to the external libraries here as DLLs, which are for the Windows platform, the exact same steps would apply to calling Frameworks under Mac OS X or Shared Libraries under Linux.

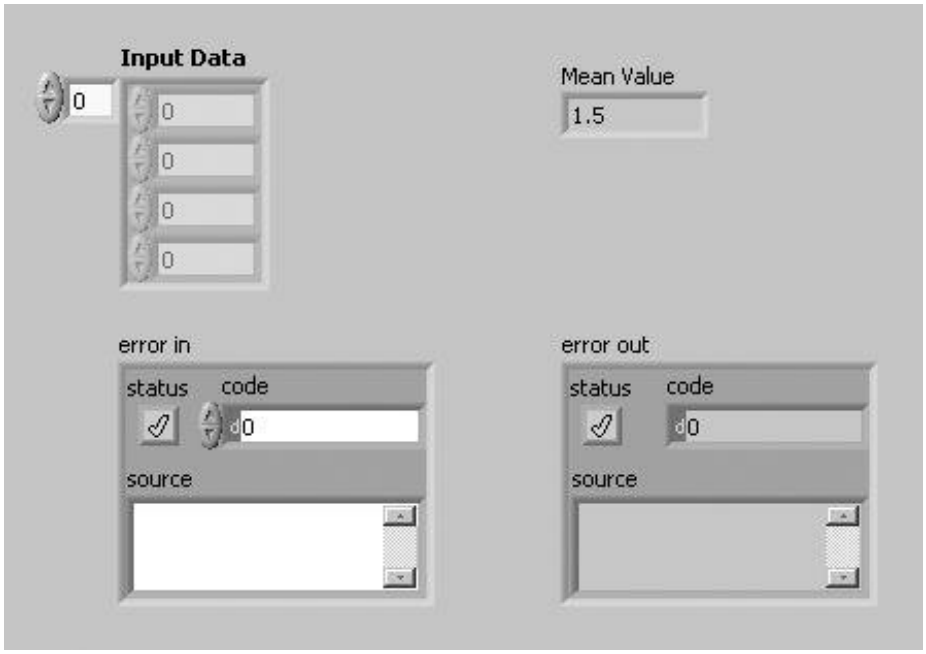
Watch Out! Calling DLLs incorrectly or calling DLLs that are buggy can make your system become unresponsive, crash LabVIEW, or even crash your entire computer. Be sure you've saved your work and test carefully when using DLLs. And in some cases, keep the fire extinguisher nearby!

Activity 14-4: Calling a DLL in LabVIEW

In this activity, you'll see how to call a DLL. To use a simple DLL that you already have on your system to compute the mean value of an array, we'll call a math-analysis DLL that already ships with LabVIEW, the *lvanlys.dll*. This DLL is actually used by many of LabVIEW's math functions.

1. Create a VI like the one shown in Figure 14.40, with an input array of numerics and an output called "mean." Save your VI as Calling a DLL.vi.

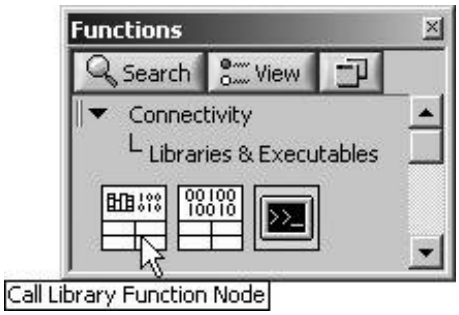
Figure 14.40. Calling a DLL.vi front panel



2. Now, locate the DLL lvanlys.dll located at `resource\lvanlys.dll` beneath your LabVIEW installation.

3. Open a new VI and put a Call Library Function Node on the block diagram (from the Connectivity>>Libraries & Executables palette).

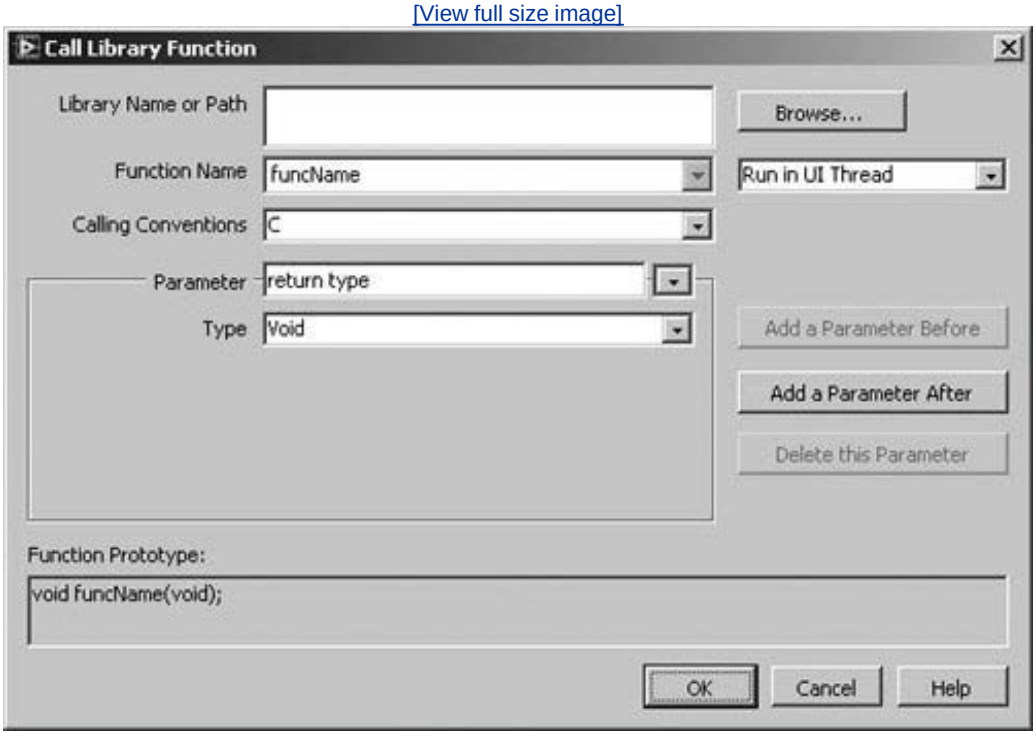
Figure 14.41. The Call Library Function Node on the palette



Right-click on the Call Library Function Node and select Configure . . . from the pop-up menu. You'll get a window like the one shown in Figure 14.42.

Figure 14.42. Call Library Function configuration window

[View full size image]



4. This is where you define what function you wish to call from the DLL, and what your input and output parameters are. Follow these steps to configure the DLL:

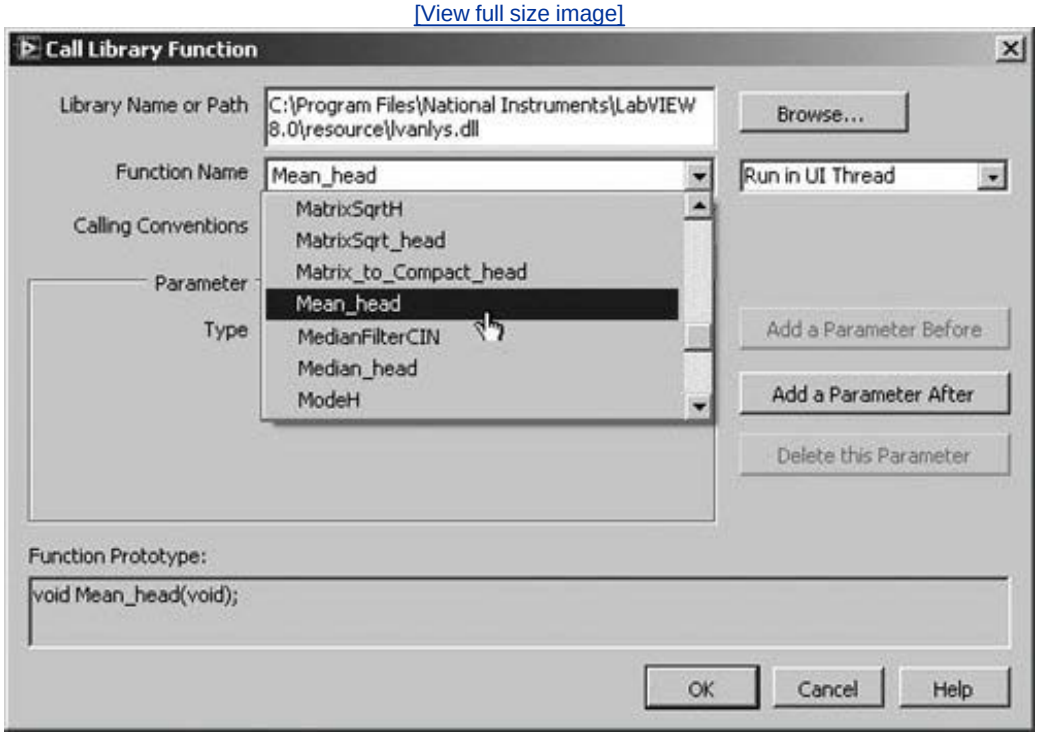
   a. First, tell it which DLL we are using. Click on the "Browse . . ." button and navigate to `resource\lvanlys.dll`, which you located earlier. Note how the path to the DLL shows up in the "Library or Path Name" field.

   b. Now click on the "Function Name" field. You'll see this DLL has LOTS of functions. We're going to use a function called Mean_head. Scroll down the list and select "Mean_head"

as your function name (see Figure 14.43).

Figure 14.43. Selecting the function name in the DLL

[View full size image]



c. Now that you've selected your function, you'll need to know what kind of inputs and outputs it expects. This is why it is essential for you to have proper documentation on your DLLs, because unlike functions in LabVIEW, the DLL functions here don't automatically show you what inputs and outputs are expected. You need to at least know what the *function prototype* looks like. So, here is your documentation for the function `Mean_head`:

The function `Mean_head` has the following function prototype in C:

```
long Mean_head(void *xH, void *mean, void *dsperr);
```

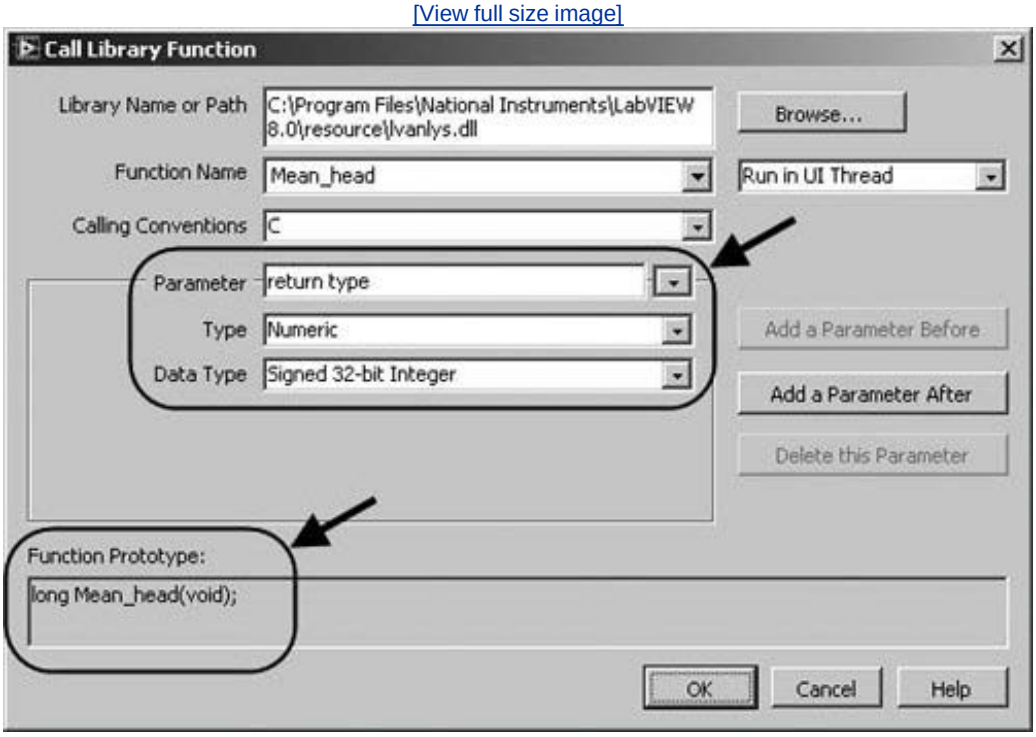
If you aren't familiar with C language, this means that the function Mean_head returns a *long* (32-bit) numeric, and takes in as arguments a pointer called `xH`, a pointer called `mean`, and a pointer called `dsperr`.

This function takes an array of numerics referenced by the pointer `xH` and calculates the mean, placing its value in the memory location referenced by the pointer `mean`. If there is an error, its numeric code is referenced by the pointer `dsperr`.

d. So your first step is to define the Return Type parameter. Looking at the previous documentation, you can see that this function returns a *long* numeric. Click on the Parameter box and choose "return type." From the "Type" box below, choose "Numeric." When you do so, a new option appears, called "Data Type." Make sure it is set to "Signed 32-bit Integer." It should look like Figure 14.44.

Figure 14.44. Setting the "return type" parameter (return value) on the DLL function

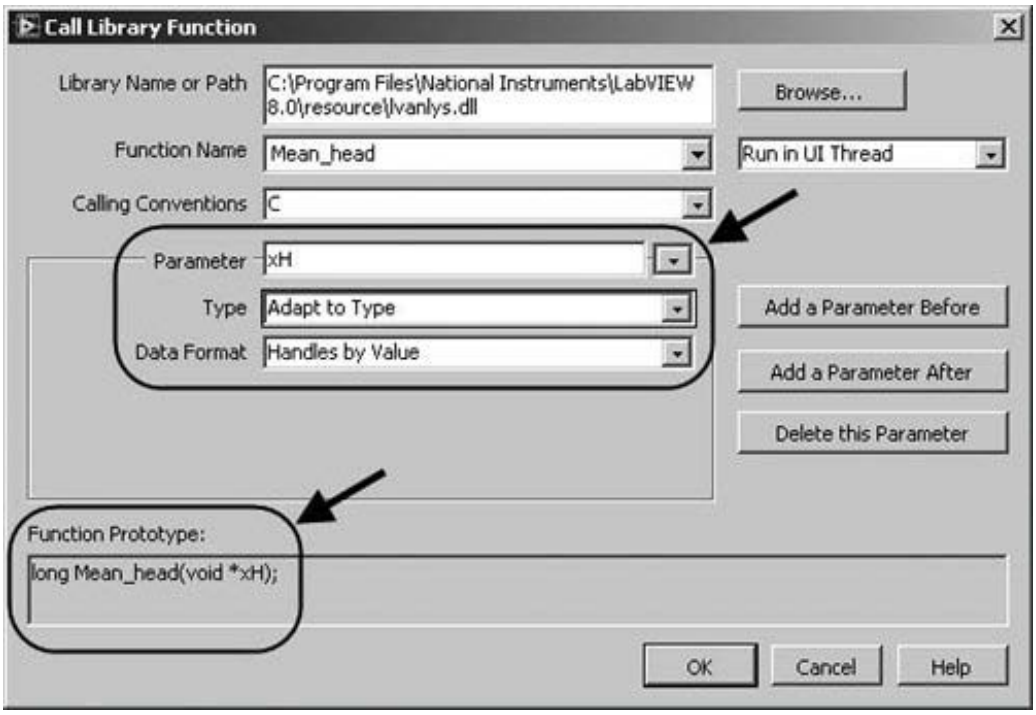
[View full size image]



e. Now that you've set the return type, it's time to define the arguments, or input parameters to the DLL function "Mean_head." Looking at the documentation, we see that there are three arguments: *xH, *mean, and *dsperr. These need to be entered in the correct order. Click on the "Add a Parameter After" button on the dialog. The Parameter field will say "arg1." Change this to say "xH." The type should be "Adapt to Type," and the data format should be "Handles by Value." Notice how the Function prototype at the bottom of the dialog is reflecting what you have entered, as seen in Figure 14.45.

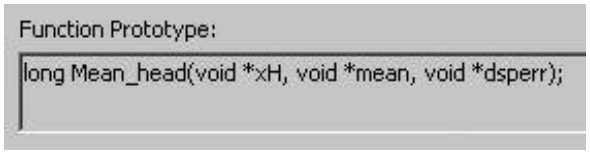
Figure 14.45. Setting the "xH" input parameter (argument) for the DLL function

[View full size image]



f. Repeat the previous step for the next two parameters: *mean and *dsperr. The "mean" and "dsperr" parameters should also be set to "Adapt to Type" and "Handles by Value." Your function prototype should look like the one in Figure 14.46.

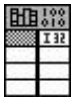
Figure 14.46. The final function prototype



Make sure your function prototype matches EXACTLY the one in Figure 14.45 and, in particular, that your parameters are in the right order.

g. Hit OK. You've configured your DLL! Your Call Library Function Node function will now look like Figure 14.47.

Figure 14.47. Your Call Library Function Node after you have configured it

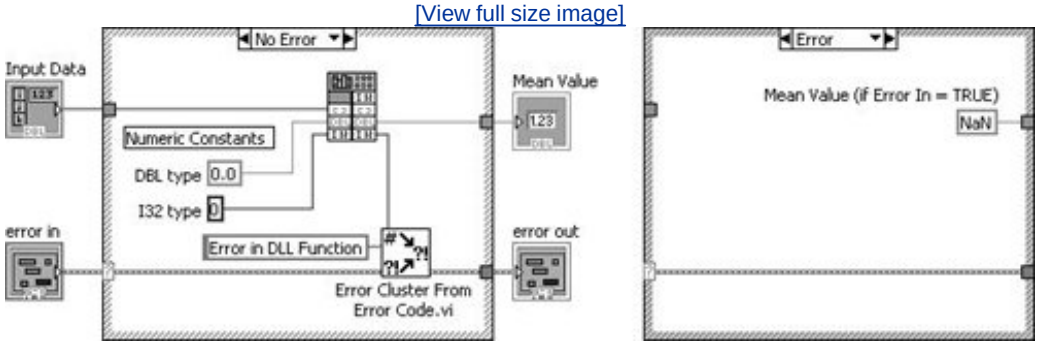


h. Notice you have three input terminals and four output terminals. The three input terminals are your arguments, and the four output terminals are the return type plus the pointer data from each of the three inputs.

i. Now wire the VI's block diagram as shown in Figure 14.48. You have to provide "dummy" inputs to the arguments, even if they represent outputs, so that LabVIEW knows what kind of data type you are using (due to the fact that these parameter types were configured as "Adapt to Type"). To do this, wire numeric constants into the input, as shown in Figure 14.48. Make sure to set the second input parameter's numeric constant to a DBL type by popping up on the numeric constant and choosing Representation>>DBL.

Figure 14.48. Calling a DLL.vi block diagram

[View full size image]



And, make sure the last input parameter is an I32 numeric by selecting Representation>>I32 from its pop-up menu.

Error Cluster From Error Code.vi

Use Error Cluster From Error Code.vi (Programming>>Dialog & User Interface palette) to convert the error or warning code to an error cluster. This VI is useful when you receive a return value from a DLL call or when you return user-defined error codes.

If `error in` contains an error when the VI is run, then the "Error" case of the Case Structure will execute. Note that we pass out *Not a Number* (NaN) in this case. It is a common convention to

output Not a Number out of a mathematical function when an error occurs.

j. Now type a few values on the front panel's `Input Data` array, and run the VI to see it compute the mean.

If all of this seems complicated for such a simple function, it is! Using pure LabVIEW is easier and much more fun. That's why you should try to avoid DLLs unless you have a good reason to use them, such as needing a routine only available as a DLL or in some speed-critical application.

There are many other complex issues involved in LabVIEW's communicating with external code. Because many of these are highly dependent on the processor, operating system, and compiler you're using, we won't attempt to go any further into discussing DLLs or CIN functions. If you'd like more details, consult the LabVIEW help documentation and the application notes at http://zone.ni.com.

PREV NEXT

Fitting Square Pegs into Round Holes: Advanced Conversions and Typecasting

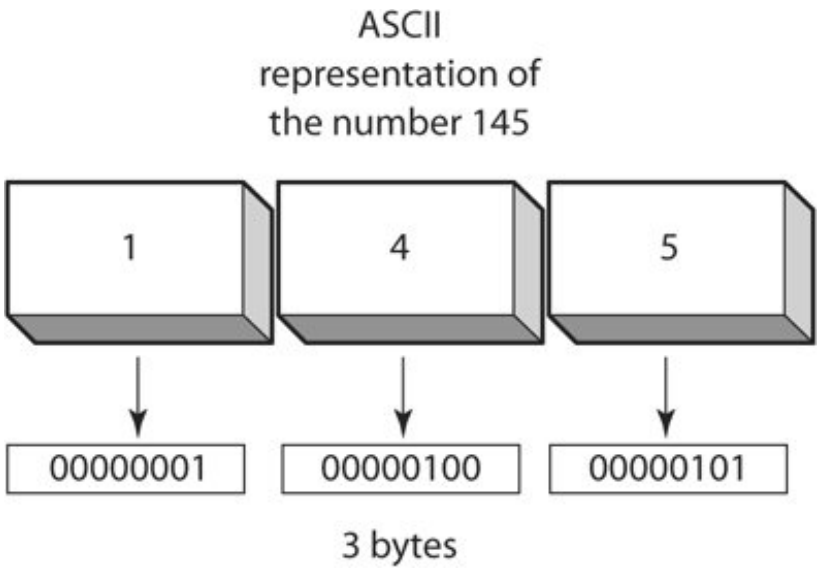
Remember *polymorphism*, discussed earlier in this book? It's one of LabVIEW's best features, allowing you to mix data types in most functions without even thinking about it. (Any compiler for a traditional programming language would scream at you if you tried something like adding a constant directly to an arraybut not LabVIEW.) LabVIEW normally takes care of doing conversions internally when it encounters an input of a different but compatible data type at a function.

If you're going to develop an application that incorporates instrument control, interapplication communication, or networking, chances are you'll be using mostly string data. Often you'll need to convert your numeric data, such as an array of floating-point numbers, to a string. We talked about this a little in Chapter 9. It's important to distinguish now between two kinds of data strings: *ASCII strings* (the sequences people readily understand, such as the characters on this page) and *binary strings* (the sequences of bytes [each containing 8 bits] in a computer's memory or disk).

ASCII strings use a separate character (a whole byte) to represent each digit in a number. Thus, the number 145, converted to an ASCII string, consists of the characters "1," "4," and "5."[1] For multiple numbers (as in arrays), a delimiter such as the <space> character is also used. This kind of string representation for numbers is very common in GPIB instrument control, for example (see Figure 14.49).

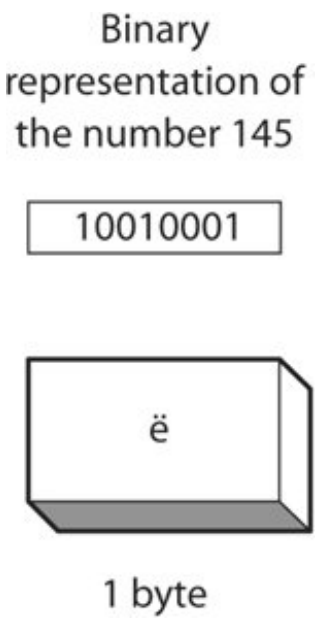
[1] Unicode, the increasingly popular alternative to ASCII, uses 4 bytes (32 bits) to represent characters, allowing a much broader range of international characters and symbols. At the time of press, LabVIEW did not have support for Unicode.

Figure 14.49. ASCII representation of 145



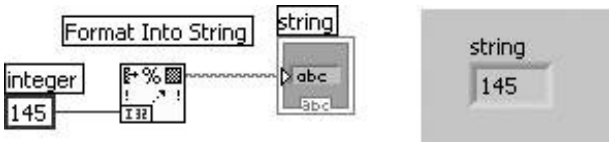
Binary strings are a bit more subtlefor one thing, you can't tell what data they represent just by reading them as ASCII characters. The actual bit pattern (binary representation) of a number is used to represent it. Thus, the number 145 in I8 representation is just a single byte in a binary string (which, incidentally, corresponds on my computer's ASCII set to the "ë" characternot that most humans could tell that means 145). Binary strings are common in applications where a minimal overhead is desired, because generally it is much faster to convert data to binary strings and they take up less memory (see Figure 14.50).

Figure 14.50. Binary representation of 145



Ultimately, all data in computers is stored as binary numbers. So how does LabVIEW know if the data is a string, Boolean, double-precision floating-point number, or an array of integers? All data in LabVIEW has two components: the *data* itself, and the data's *type descriptor*. The data type descriptor is an array of I16 integers that constitute code that identifies the representation of the data (integer, string, double-precision, Boolean, etc.). This type descriptor contains information about the length (in bytes) of the data, plus additional information about the type and structure of the data. For a list of data type codes, read the Fundamentals>>How LabVIEW Stores Data in Memory section of the LabVIEW Help. In a typical conversion function, such as changing a number to a decimal string, the type descriptor is changed and the data is modified in some way. The common conversion functions you use most of the time convert numbers to ASCII strings, such as in this example (see Figure 14.51).

Figure 14.51. Converting a number to an ASCII string

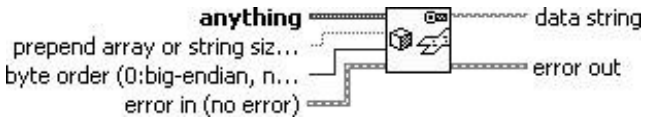


The topics in the rest of this section can be very confusing to beginners. If you do not have any need to use binary string conversions in your application, you can safely skip the rest of this section.

In some cases, you may want to convert data to a *binary* string. Binary strings take up less memory, are sometimes faster to work with, and may be required by your system (such as a TCP/IP command, or an instrument command). LabVIEW provides a way to convert data to binary strings using the Flatten To String function. You can access this function from the Programming>>Numeric>>Data Manipulation palette.

The anything input to Flatten To String can be literally any LabVIEW data type, including complex types such as clusters (see Figure 14.52). The function returns a single output: the binary data string representing the data. You can also choose the byte order (big endian or little endian) and whether to *prepend array or string size*.

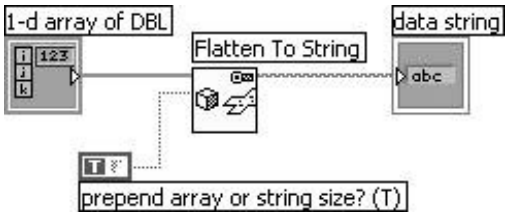
Figure 14.52. Flatten To String



When you flatten a string or array, the default behavior of Flatten To String is to prepend the string or array size to the front of the data string (see Figure 14.53). If you do not want

the size information, then set the prepend array or string size? input to FALSE.

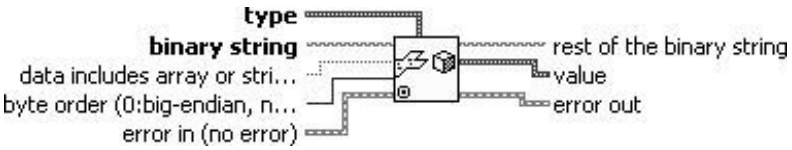
Figure 14.53. Using the prepend array or string size argument when calling Flatten To String



The key to understanding these confusing binary flattened strings is to realize that most of the time you don't need to manipulate or view the strings directlythey are not intended to be human readable. They are just a convenient, compact, and generic format for passing data to a file, a computer over a network, an instrument, and so on. Later, to read the data, you just need to unflatten the string back into the appropriate LabVIEW data type.

To read back binary strings, use the inverse function, Unflatten From String (see Figure 14.54). The type input is a dummy input for specifying (by wiring to it) the data type that the binary string represents. The value output will contain the data in the binary string, but will be of the same data type as type. An error will be generated (passed out of error out) if the conversion is unsuccessful. Just like the Flatten To String function, you can specify the byte order (big endian or little endian) of the binary string.

Figure 14.54. Unflatten From String



The byte order and prepend settings passed to Unflatten From String must match those passed to Flatten To String when the data was originally flattened. Otherwise, the data will not be unflattened properly.

Here's the example of how to convert back your flattened binary string to a DBL array.

You wire the binary string input, as well as a dummy (empty or not) 1-D array of DBL to specify the data type, and you get back your original array of DBL. Notice how you flattened and unflattened without ever needing to "peek" or "manipulate" the binary strings directly.

For very fast, efficient conversions, let's take a final look at another, very powerful LabVIEW function: Type Cast (see Figure 14.56).

Figure 14.55. Calling Unflatten From String to convert an array stored as a flattened string back into an array

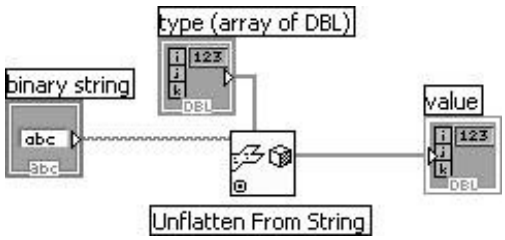
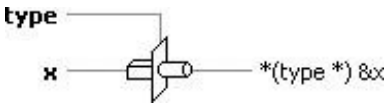


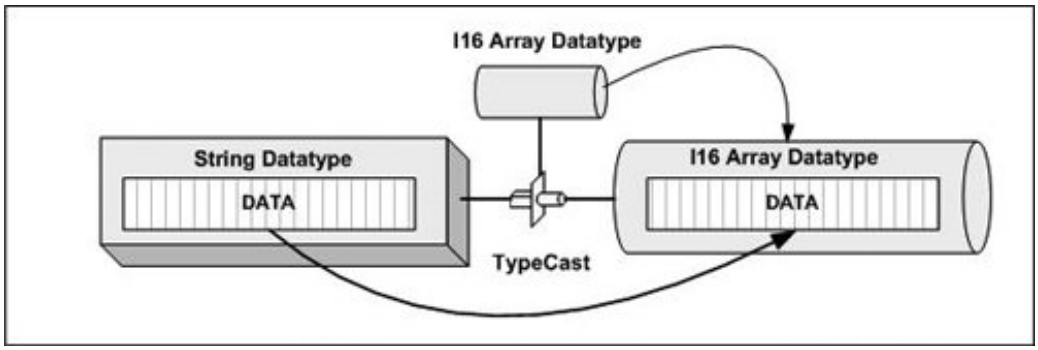
Figure 14.56. Type Cast



This function allows you to change the data type descriptor, without changing the data at all. There is no conversion of the data itself in any way, just of its type descriptor. You can take almost any type of data (strings, Booleans, numbers, clusters, and arrays), and call it anything else. One advantage of using this function is that it saves memory because it doesn't create another copy of the data in memory, like the other conversion functions do. Figure 14.57 shows an illustration of how the Type Cast operation changes the wire's type but leaves the data unchanged.

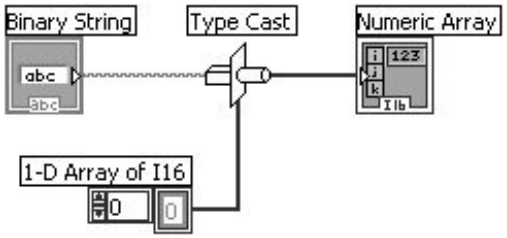
Figure 14.57. The Type Cast function only changes the wire type, not the data flowing through the wire

[View full size image]



A common use of type casting is shown in Figure 14.58, where some instrument returns a series of readings as a binary string, and they need to be manipulated as an array of numbers. We are assuming that we know ahead of time the binary string is representing an array of I16 integers.

Figure 14.58. Type casting a binary string to an array of I16 integers



Type casting from scalars or arrays of scalars to a string works like the Flatten To String function, but it has the distinct feature that it doesn't add or remove any header information, unlike the binary strings created by the Flatten and Unflatten functions. The type input on the Type Cast function is strictly a "dummy" variable used to define the typeany actual data in this variable is ignored.

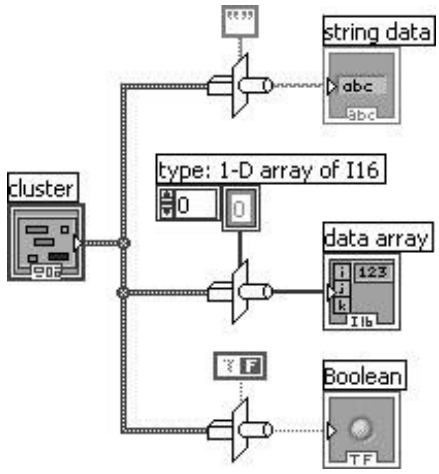
You need to be very careful with this function, however, because you have to understand exactly how the data is represented so your type casting yields a meaningful result. As mentioned previously, binary strings often contain header information (such as the output from the Flatten To String function). Type Cast does not add or remove any header information in binary strings. If you don't remove headers yourself, they will be interpreted as data and you'll get garbage as a result.

Make sure you understand how data to be type cast is represented. Type casting does not

do any kind of conversion or error-checking on the data. Byte ordering (MSB first or last) is platform-dependent, so understand what order your data is in.

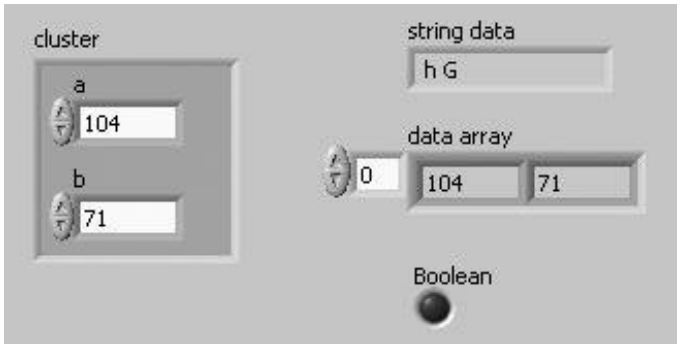
You might try experimenting with the Type Cast function to see what kind of results you get. Take a look at Type Casting.vi, found in the `EVERYONE\CH14` folder on the CD-ROM. To show you how bizarre you can get with type casting, we took a cluster containing numeric controls (type I16) and cast the data to a string, a numerical array, and a Boolean (see Figure 14.59).

Figure 14.59. Type Casting.vi block diagram showing the Type Cast function converting a cluster to many different data types



The front panel (see Figure 14.60) shows you what results we obtained with two numbers.

Figure 14.60. Type Casting.vi front panel showing the results of type casting a cluster to many different data types



The string returned two characters corresponding to the ASCII values of 104 and 71. The array simply re-organized the numbers into array elements instead of cluster elements. And the Boolean? How would a Boolean variable interpret cluster data? Because we didn't know, we tried this, and found out the Boolean is TRUE if the first byte of the raw data is 255; otherwise, it's FALSE. So in this case, if the first element of the cluster has a value greater than 255, the Boolean is TRUE; otherwise, it's FALSE. Pretty strange, eh?

PREV NEXT

You Can Be Anything: Variants

In LabVIEW, you cannot connect wires of different types, unless the data source can be coerced to the data sink. For example, you *cannot* wire a string control to a DBL numeric indicator, because a string cannot be coerced to a DBL. However, you *can* wire an I32 numeric control to a DBL numeric indicator, because an I32 can be coerced to a DBL. This requirement that the data source must match or be coercible to the data sink is one that we take for granted. But, it is extremely important because it ensures that our VI runs properly. You certainly would not want LabVIEW to allow you to wire both a string data type and a numeric data type to an Add function (which makes no sense), and simply give you an error when the Add function executesthat would just waste your time. In general, you need to know about these type mismatch problems while you are *editing* your VI rather than when we are *running* your VI. LabVIEW's strict typing is one of its main advantages.

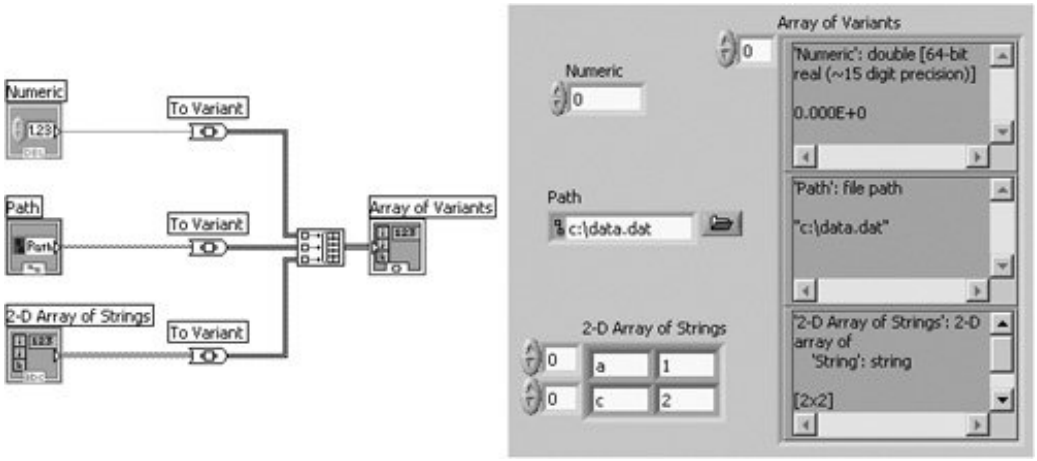
Sometimes, though, you may desire flexibility while wiring. You want to be able to wire different types (or possibly *any* type) because the particular code you are writing is required to handle many types of data (or possibly *any* type of data). For example, if you are writing a software component that routes messages between different parts of large software applications, you probably don't care about the type of data that is in the messageit's not your job to care about the *type of data* you are routing; it's your responsibility to know how to *route the data* to its destination.

LabVIEW has a very special data type that provides the flexibility and features needed for just this sort of situation: the *variant* data type. If you have programmed in Visual Basic before, you're already familiar with this "universal" data type.

Figure 14.61 shows three dissimilar data types converted to variants using the To Variant function (Programming>>Cluster & Variant palette) and then built into an array of variants.

Figure 14.61. Creating an array of variants allows you to create an array of mixed types (different data types inside each variant)

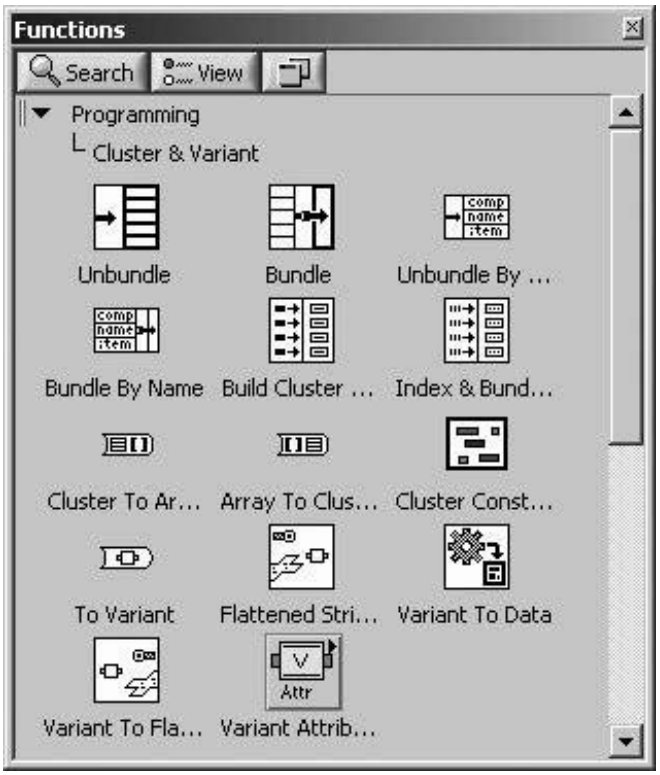
[View full size image]



On the front panel, a variant control (found on the Modern>>Variant subpalette of the Controls palette) can display both the type and (simple) textual representation of the data. You can toggle the visibility of the data and type information from a variant's pop-up menu options Show Data and Show Type.

The functions for operating on variant data are found on the bottom two rows of the Programming>>Cluster & Variant palette (see Figure 14.62).

Figure 14.62. Cluster & Variant palette

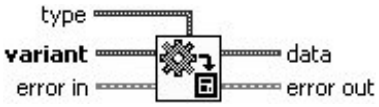


To Variant (Programming>>Cluster & Variant palette) converts any LabVIEW data to variant data (see Figure 14.63).

Figure 14.63. To Variant

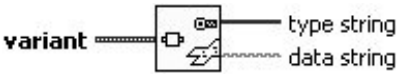
Variant To Data (Programming>>Cluster & Variant palette) converts variant data to a specific LabVIEW data type so LabVIEW can display or process the data (see Figure 14.64).

Figure 14.64. Variant To Data



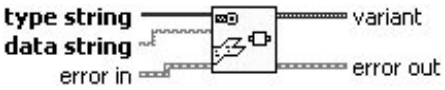
Variant To Flattened String (Programming>>Cluster & Variant palette) converts variant data into a flattened string and an array of integers that represent the data type (see Figure 14.65).

Figure 14.65. Variant To Flattened String



Flattened String To Variant (Programming>>Cluster & Variant palette) converts flattened data into variant data (see Figure 14.66).

Figure 14.66. Flattened String To Variant



These functions allow you to convert LabVIEW data to variants and variants to LabVIEW data, but

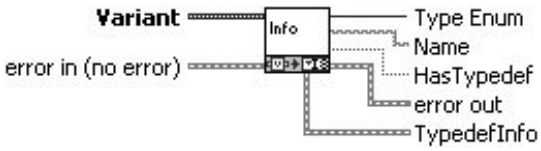
nothing more. They do not provide you with a means to inspect variant data, nor operate on it.

There are a limited set of tools that ship with LabVIEW for inspecting and operating on variant data. These are sort of "hidden" in LabVIEW (they are not found in the Functions palette), but you can find them in the following folder: `vi.lib\Utility\VariantDataType`.

Probably the most useful of these functions is GetTypeInfo.vi, which tells you both the type and name of variant data. This is very useful, because once you know the type of a variant, then you can use the Variant To Data function to convert it back into that type.

Get Type Info.vi (`vi.lib\Utility\VariantDataType` folder) returns information about the data type stored in a variant, including the data type (as an enum), the data name (as a string), and typedef info (see Figure 14.67).

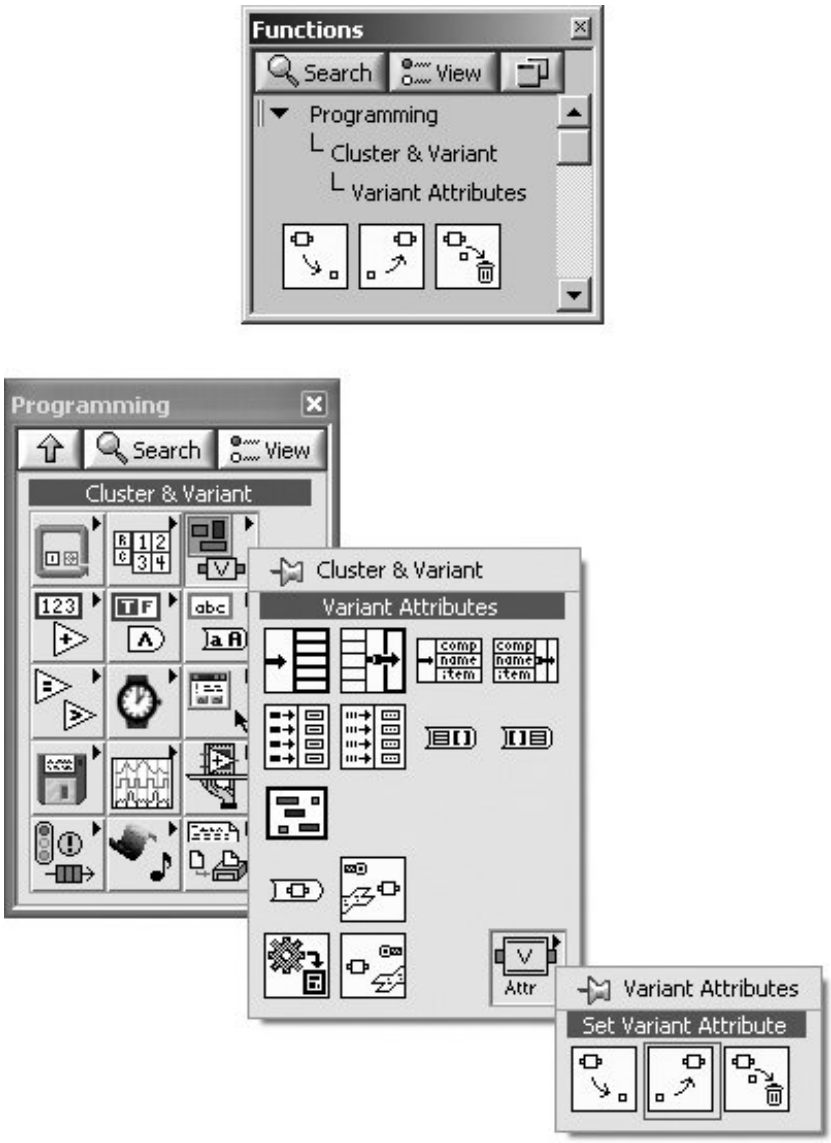
Figure 14.67. Get Type Info.vi



Finally, one other useful feature of variants is their capacity to store additional information in the form of Variant Attributes. Variant attributes are key-value pairs of data that can contain any sort of information you wish. For example, if you want to know the time when a piece of data was created, you could store a "Time" attribute in the variant. The attributes of a variant can be any kind of data type and contain anything you want.

The Variant Attribute palette shown in Figure 14.68 has functions for reading, setting, and deleting variant attributes. The name of the attribute is a string, and the value of a variant attribute is . . . a variant itself!

Figure 14.68. Variant Attributes palette

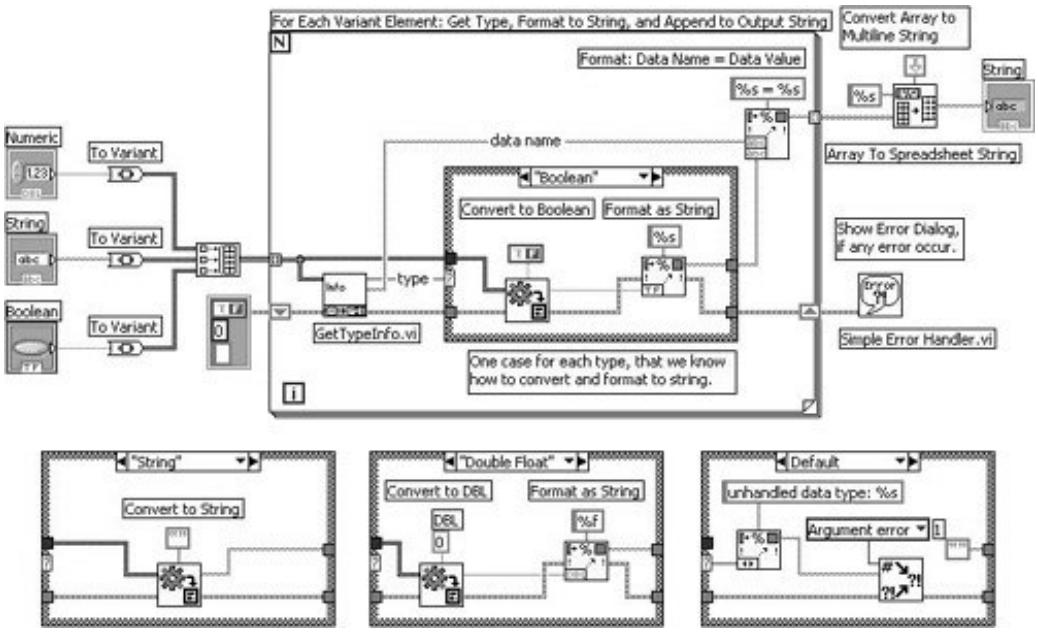


Using Variants: Creating Generic Software Components

Let's look at an example that highlights the power and flexibility of variants. Figure 14.69 is a VI that formats an array of variants (anything) into a string of key-value pairs, like you might see in an configuration file (INI) file.

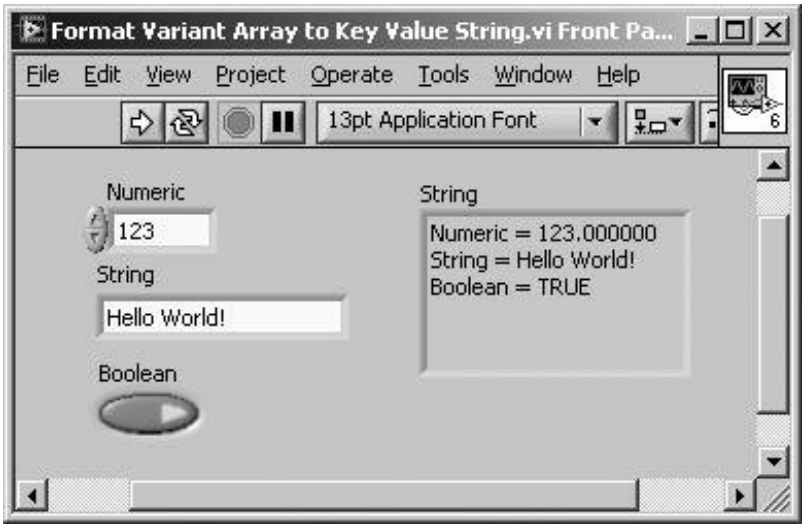
Figure 14.69. Block diagram of a VI that converts an array of variants into a multi-line string of key-value pairs

[View full size image]



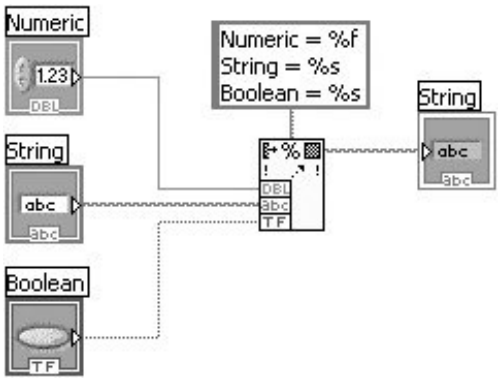
This end result is shown in Figure 14.70.

Figure 14.70. Front panel of a VI that converts an array of variants into a multi-line string of key-value pairs



You might be thinking to yourself that we have just done a whole lot of work in order to do something as simple as formatting a string from the values of three controls. Couldn't we simply whip up a VI that looks like Figure 14.71?

Figure 14.71. Block diagram showing an "easy" way to create a string of key-value pairs using Format Into String



Sure we could, and we would save a lot of time now. But, our variant implementation will save us time in the future. Here is how:

- The variant implementation gets the data names programmatically from the variant data. In our "simple" example, we have to change the name in two locations. This is extra work, and we might make a mistake!
- In our simple example, if we want to change the equal sign (=) to a colon (:), or another character, we have to do it once for every data element. This change requires work that is proportional to the number of elements.
- Our simple example is limited to the three elements that are wired up to it. But, our variant example can easily handle any number of elements in the variant array. It could even be converted into a subVI (with an array of variants as an input parameter) and used in multiple locations of our code or even in multiple projects!

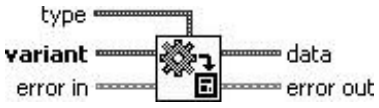
The moral of the story is that things that appear simple at first, might not be the best scaleable solution. By using variants, you *can* create scaleable, generic, powerful, and flexible tools.

If you are interested in doing much work with variants, you will benefit from a powerful set of tools for parsing type descriptors and flattened data. Just such a set of tools is available from OpenG.org. See Appendix C, "Open Source Tools for LabVIEW: OpenG," for more information.

ActiveX and the Variant Data Type

ActiveX is a Microsoft technology that allows you to reuse *components* of code in other applications. For example, a component could be the ADO object, which allows you to access almost any kind of database, or the Windows Media Player component, which allows you to play audio and video files. Even LabVIEW itself can become an ActiveX component. ActiveX is a fairly advanced topic, and we'll talk more about it in Chapter 16.

Figure 14.72. Variant to Data function



When exchanging data with ActiveX servers and controls, you will often pass (and be passed) data as a variant. However, *data passed from ActiveX into LabVIEW is passed as an ActiveX variant and not a true LabVIEW variant* (although the wire pattern and color look exactly the same). Do not attempt to operate on the variant data using variant utility functions and VIs (for example, Variant To Flattened String will fail); but rather, immediately convert the ActiveX variant into LabVIEW data using the Variant to Data function.

That's all we will say about ActiveX variants. Please keep this in mind as you explore ActiveX in Chapter 16.

PREV NEXT

Wrap It Up!

LabVIEW gives you the capability to define and use polymorphic VIs; that is, VIs that can automatically adapt to different data types on their inputs and outputs. The advanced file IO functions in LabVIEW let you dig down into text and binary files and have random access to the data within them. You can also use .ini, or configuration files, with LabVIEW. If you need to share LabVIEW with other applications, or if you need to import external code, LabVIEW can both call and create external DLLs (Windows) or Shared Libraries (Mac OS X and Linux). Finally, we took a look at advanced data conversion through the use of the Type Cast function and the Variant functions.

Additional Activities

Activity 14-5: Read Text File by Page

Modify Read Text File Advanced.vi (from Activity 14-1) so that every time you press a `NEXT PAGE` button, the VI reads the next `PAGE_SIZE` characters from file, and when you press a `PREVIOUS PAGE` button, it reads the previous `PAGE_SIZE` characters from file. Save your VI as Read Page from Text File.vi.

- *Store the file marker (the current position) in a shift register and increment or decrement it by `PAGE SIZE` depending on whether the `NEXT PAGE` or `PREVIOUS PAGE` button is pressed.*
- *Use an Event Structure for capturing button press events.*
- *Read the file using the file marker as the start position and page size as the number of characters.*
- *Make sure that you do not read before the beginning of the file (position = 0) or past the end of the file (position = end of file + 1).*

Now, for extra (geek) points, use a Vertical Scrollbar control (which you learned about in Chapter 13, "Advanced LabVIEW Structures and Functions") to control the page that is read.

Configure the Scrollbar properties, as follows:

- *Set Doc Min to 0 (zero).*
- *Set Doc Max to the file size (total number of characters).*
- *Set Increment to 1 (one character).*
- *Set Page Size to the number of characters you want in a page.*

PREV NEXT

15. Advanced LabVIEW Features

Overview

You've learned quite a bit about LabVIEW's built-in functions, structures, and variablesnow it's time to examine the potpourri of advanced tools you can use in LabVIEW to make your programming easier and more versatile. By tweaking the numerous environment options, you can customize the appearance and functionality of LabVIEW so it works best for you. You will also learn the VI Property options; for example, how to set up your VI so you can select front panel controls through the keyboard instead of the mouse. We'll look at the powerful VI Server and how you can use it to dynamically control VIs. LabVIEW's numeric types give you the option of representing them as binary, octal, or hexadecimal, as well as decimal representation. In addition, you can include built-in units as part of a numeric variable and allow LabVIEW to perform automatic conversions. You will see how to instantly transform a section of your diagram into a subVI. Finally, LabVIEW has some useful development tools in the Tools menu, such as the Find function, which lets you quickly search for any object, subVI, or text and optionally replace one or all instances with another object, subVI, or text of your choice.

Goals

- Use the Options to customize you LabVIEW environment
- Become familiar with the VI Properties options that allow you to make choices on the appearance and execution of your VI
- Be introduced to the VI Server and what it does
- Learn about VI references and control references
- Become familiar with some properties and methods of the Application class, the VI class, and the Control class
- Understand what a reentrant VI is
- Be able to set up keyboard access of controls
- Learn how to represent numbers with other radices and assign them units
- Create a subVI from a selection of the block diagram
- Discover the utilities under the Tools menu
- Use and create custom probes
- Learn how to Find and Replace objects or text in your VIs

Key Terms

- Options
- VI properties
- VI Server
- Application class
- VI class
- Control class
- Methods and properties
- Invoke node
- Property node
- Priority
- Reentrant execution
- Key focus
- Radix
- Unit
- Find
- Profile window
- History
- Custom probe
- VI hierarchy

> *"The time has come," the Walrus said,*
> *"To talk of many things:*
> *Of shoesand shipsand sealing-wax*
> *Of cabbagesand kings*
> *And why the sea is boiling hot*
> *And whether pigs have wings."*

Lewis Carroll (183298), English writer, mathematician *Through the Looking Glass,* ch. 4 (1872)

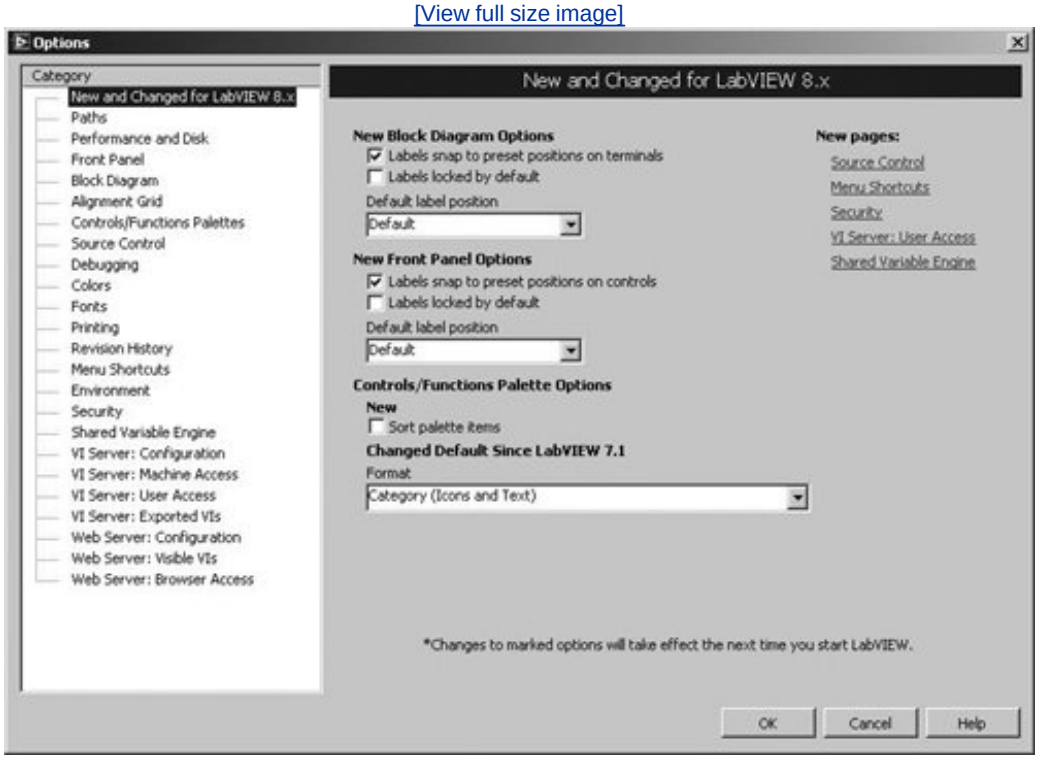
Exploring Your Options: The LabVIEW Options Dialog

Maybe you've already used the Options . . . command under the Tools menu. In any case, it's a good idea to skim through all the options available, because often here is where you will find the solution to that persistent LabVIEW problem you've had for a month. You can specify things such as search paths, default front panel colors and fonts, time and date formats, and so on.

Figure 15.1 shows some of the option categories.

Figure 15.1. LabVIEW Options dialog

[View full size image]



Each of the categories shown in the previous figure brings up several options when selected. Although a comprehensive preference list would be too long to show here, some preference options are more commonly used and turn out to be quite useful:

- The New and Changed for LabVIEW 8.x category lists all the options that are new or have changed in the latest version of LabVIEW. This makes it easy to learn about what has changed. These options are also accessible from their respective categories.

- In the Paths options, you can specify LabVIEW's default directory. Normally, this is just the `LabVIEW` directory. However, if you keep your project VIs somewhere else, you can change the default path so that when you start LabVIEW, it's easier to open your project VIs. Generally, you will not need to change these settings.

- The Front Panel options, shown in Figure 15.2, control some important settings. For example, "Use smooth updates during drawing" turns off the annoying flickering you see on a graph or chart that is updated regularly. "Use transparent name labels" is another popular option because many people don't like the 3D box on the control and indicator labels. Finally, you can set the blink speed here (for the blink attribute of an object)you'd think this feature would be a programmatic option, but it's buried here instead.

Figure 15.2. LabVIEW Options for Front Panel appearance and behavior

- In the Alignment Grid options, you can display or hide the front panel and block diagram grid lines, and enable or disable the automatic grid alignment feature.

- The Colors preferences let you set the default colors for the front panel, block diagram, blink colors, and so on. If you just must have that magenta background all the time, this is the place to set it.

- The Printing preferences allow you to choose between bitmap, standard, and Postscript printing. Depending on the printer you have, you may need to choose a different option to get

the best print results.

- The Environment preferences allow you to choose things like whether you want the "Getting Started" window to come up when you launch LabVIEW, whether you want LabVIEW to offer "just-in-time advice," and many other tweaks.

Take a look around at the rest of the categories. You can specify font styles, time and date styles, levels of resource sharing between LabVIEW and other applications, security options, and so on. You'll notice also there are options for the VI Server, which we'll talk about shortly, and the Web Server, which is discussed in Chapter 14, "Advanced LabVIEW Data Concepts."

PREV NEXT

Configuring Your VI

Often you may want your program to bring up a new window when a certain button is pressed or a certain event occurs. For example, you may have a "main" front panel with several buttons that present options. Pressing one of the buttons would lead the user to a new screen, which may in turn contain an "Exit" button that would close the window and return to the main front panel. Because this pop-up window will usually be the front panel of a subVI, you also may want to customize what features of this window should appear, such as the Toolbar, or the clickable box that closes the window.

You can set several options that govern the window appearance and the execution of your VIs in two different places:

1. VI Properties . . . (from the pop-up menu of the VI's icon pane, which you will find in the upper-right corner of the front panel window).

2. SubVI Node Setup . . . (from the pop-up menu on a subVI's icon in the block diagram).

A very important distinction should be made at this point: the setup options under SubVI Node Setup for a subVI affect *only that particular instance of the called subVI*, while the VI Setup options *always* take effect, whether it's run as a top-level VI or called as a subVI. We'll start first with the few and simple SubVI Node Setup options.

It is best to make your settings in the VI Properties . . . dialog rather than the SubVI Node Setup . . . dialog. The reason for this is that it is not obvious that a SubVI has been configured using the SubVI Node Setup . . . dialog. It can take a long time to track down why different instances of SubVIs are behaving differently, and the SubVI Node Setup . . . dialog is the last place people usually look.

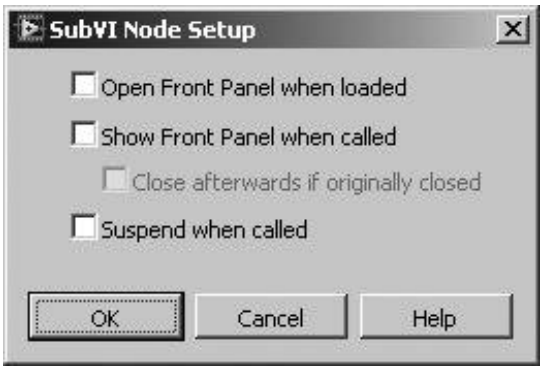
SubVI Node Setup Options (Are Evil)

As we just mentioned in the previous warning, the SubVI node setup options can cause a great many problems and they are not commonly used. (If one instance of a VI is misbehaving, the SubVI Node Setup options is the first place to look!) It is, however, important that you know how this feature worksyou might just run across a node or two that has these options set. So, we will describe the

various SubVI Node Setup . . . options and do a simple activity, now that you have had a stern warning not to use them.

When you select the setup option from the pop-up menu on a subVI, you get the dialog box shown in Figure 15.3.

Figure 15.3. SubVI Node Setup dialog



You can select any of the following options:

- Open Front Panel when loaded The subVI's front panel pops open when the VI is loaded into memory (such as when you open a VI that calls this subVI).
- Show Front Panel when called The VI front panel pops open when the subVI is executed. You'll find this option and the next to be pretty useful for creating interactive VIs.
- Close afterward if originally closed Causes the subVI front panel to close when its execution is complete, giving that "pop-up" window effect. This option is only available if the previous option is selected.
- Suspend when called Same effect as setting a breakpoint; that is, causing the VI to suspend execution when the subVI is called. You can use this as a debugging tool to examine the inputs to your subVI when it is called but before it executes.

Remember, all the subVI setup options apply only to the particular subVI node you set them on. They do not affect any other nodes of the same subVI anywhere else. Oh, yes, and you should not use any of these options.

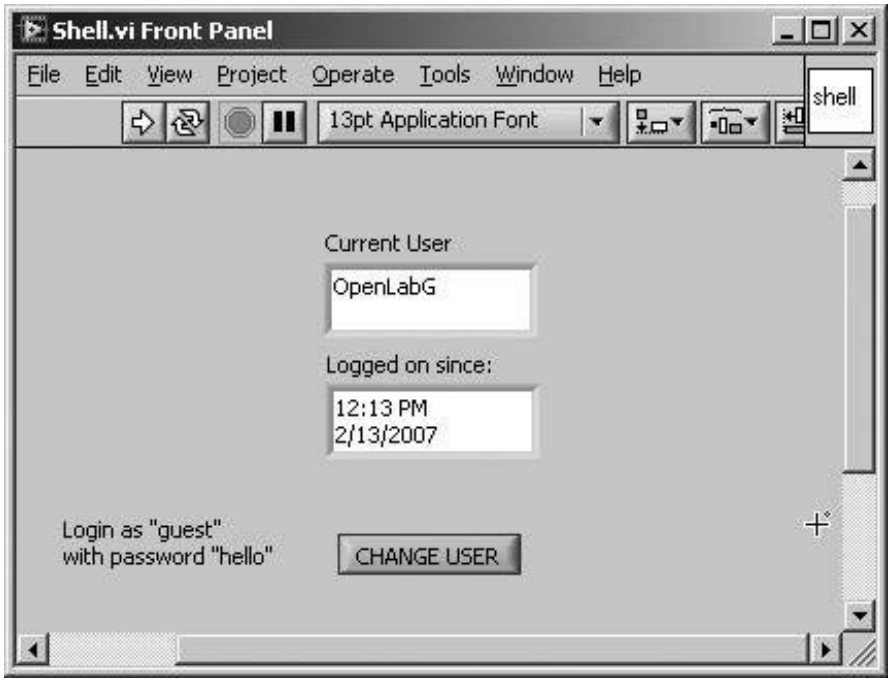
Activity 15-1: Using SubVIs

This activity will let you use the SubVI Setup Options to create a "login" shell that can be used with any application.

1. Make a simple front panel, as shown in Figure 15.4, that will call a "pop-up" subVI when the

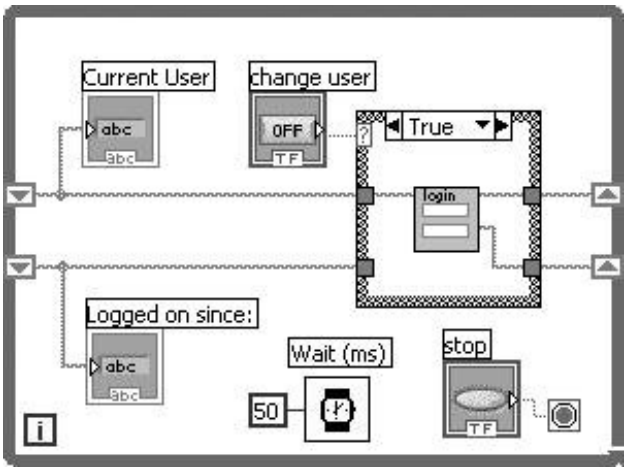
button `Change User` is pressed. Call this top-level VI Shell.vi.

Figure 15.4. Front panel of the VI you will create during this activity



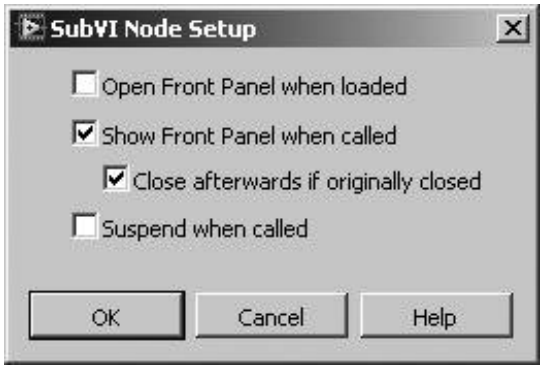
2. Use the subVI Login.vi (which you will find on the CD) to build the block diagram, as shown in Figure 15.5.

Figure 15.5. Block diagram of the VI you will create during this activity



3. Pop up on the Login subVI, and choose SubVI Node Setup. Select only the Show Front Panel when called and Close afterwards if originally closed options, as shown in Figure 15.6.

Figure 15.6. SubVI Node Setup options for the Login.vi subVI



When this VI is run, it will display the last value of the `Current user` (from the uninitialized shift register), until the Login VI is called and changes it.

Login must be closed before you run this example. Otherwise, it will not close after it finishes executinghence the phrase, "Close afterwards if originally closed."

Although you just learned how to use the SubVI Setup Options to create a subVI as a pop-up dialog, this practice is generally discouraged in any important LabVIEW application. We showed you this because you'll see it used in some older LabVIEW applications and even some of LabVIEW's examples. So it's OK to use this method for quick and trivial VIs. But for a more robust application, you will learn a better way to open and close the front panels of subVIs, using VI Property Nodes, later in this chapter.

VI Properties Options

VI properties are a bit more numerous. The dialog box that appears when you choose VI Properties . . . by popping up on the icon pane gives you a number of property sets: General, Memory Usage, Documentation, Revision History, Security, Window Size, Window Appearance, Execution, and Print Options. Some of these are read-only properties (such as memory usage); others are configurable. Let's examine these in more detail briefly.

General

This VI property set shows you the file path to the VI, the icon, and so on.

Memory Usage

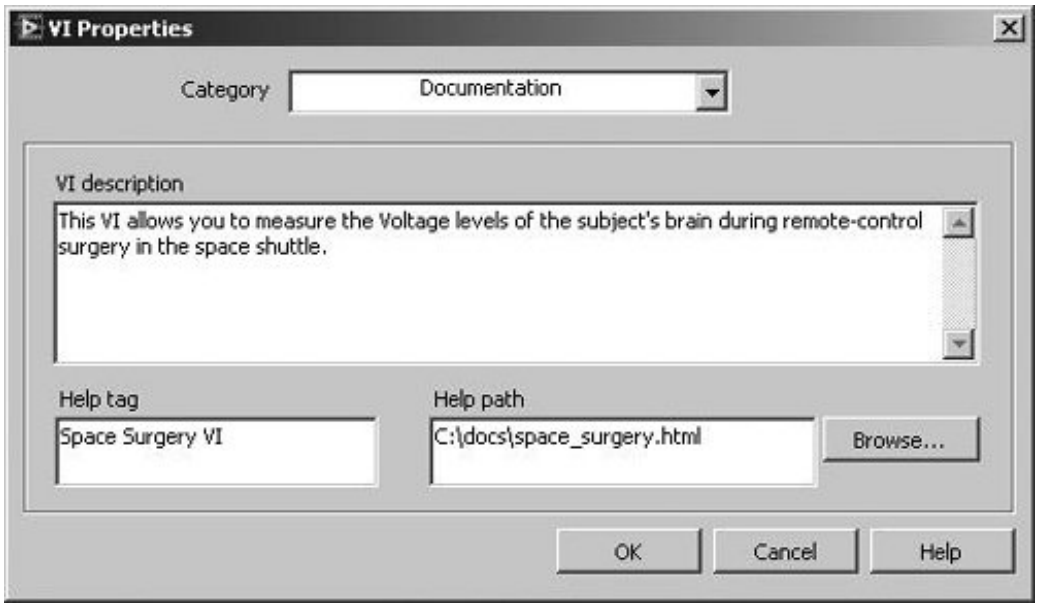
This property pane shows you how much RAM your VI is using.

Documentation

Here you can (and should) type in documentation for your VI. Good programming practices dictate that you should always type a description of your VI in the VI Description field, as shown in Figure 15.7. This description will show up in the Help window if you hover the mouse button over your VI's icon. You can also optionally enter a Help Tag and a Help Path to an external Help file.

Figure 15.7. Documentation section of the VI Properties dialog

[View full size image]



You can enter a help path to HTML (`.html`) files on all platforms or compiled help (`.chm`) files on Windows, Apple Help on Mac OS X, and QuickHelp on Linux. See Chapter 17, "The Art of LabVIEW Programming," for more information on adding online help to VIs.

Revision History

This property set allows you to set up a very simple source code history. It gives you the option of seeing what revision your VI is in, and any comments from previous revisions. This is only useful if you have been typing comments in this window every time you save the VI.

Editor Options

These options allow you to set the Alignment Grid Size and the Control Style for controls created from the right-click Create Control and Create Indicator shortcut menu options of various LabVIEW objects.

Security

By default, anyone who has your VI can view and modify the source code (the block diagram). In some cases, you may not want others to modify your block diagram code, either by accident or on purpose. Or you might not even want others to see the block diagram. Here you can specify a password that will be required for anyone to view or modify the block diagram.

If you password-protect a VI, be sure you remember or write down the password somewhere. There is NO way to access the VI's code if you forget the passwordnot even NI can help you. Be careful!

Password-protected VIs can be run on all platforms (Mac OS X, Windows, and Linux) and newer LabVIEW versions than where the VIs were created. However, VIs that have had their block diagrams removed can only be run on the version of LabVIEW they were created in. So, password protection of VIs might be a good way for you to distribute your VIs to others who may need to open them on different platforms or newer LabVIEW versionsthis allows you to distribute a single version of your files.

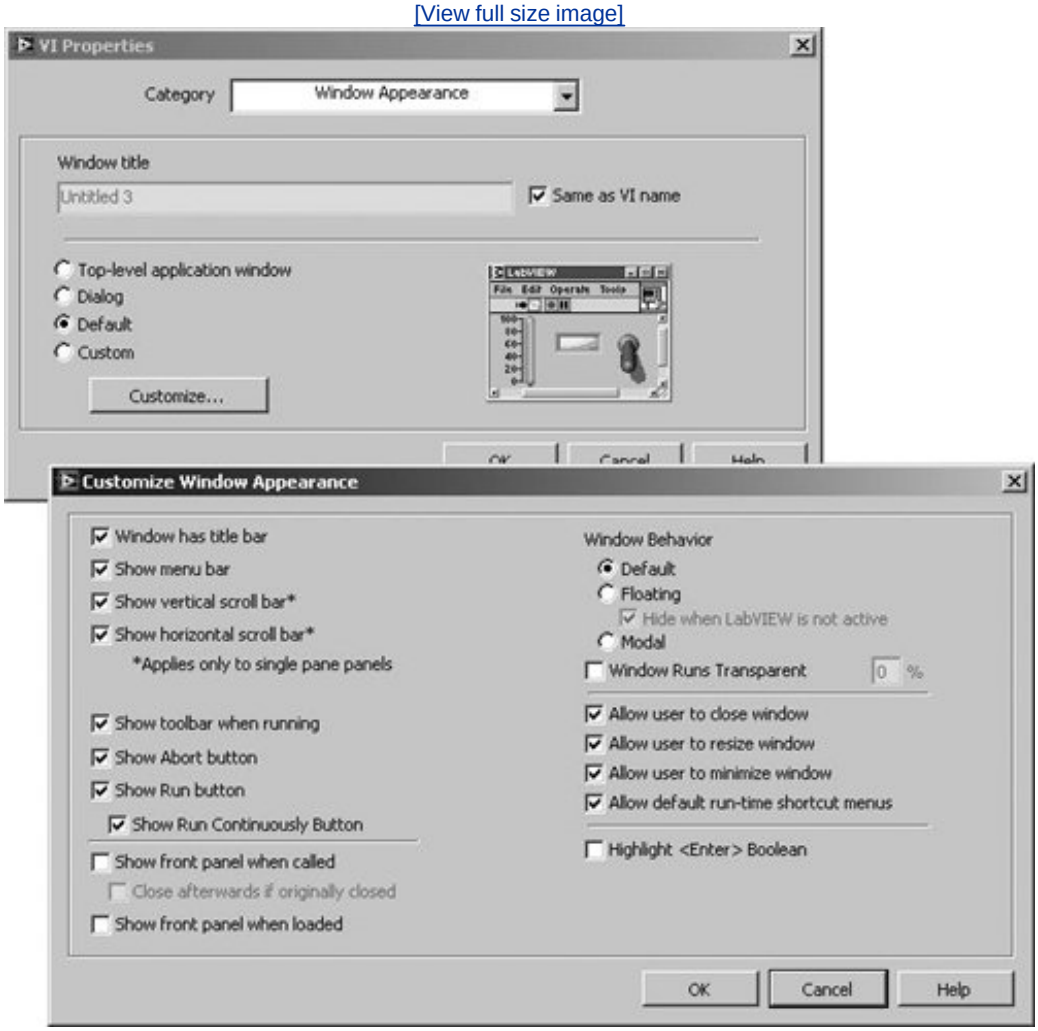
If you choose to distribute password-protected VIs, it is a good idea to only password protect a copy of the VIsnot your original, project source VIs. You might also consider selecting a long, random password, because nobody will need to get into the VIs (you have the original VIs that are not password protected). You can create a Source Distribution project Build Specification (in the LabVIEW Project Explorer) or you can use the OpenG Builder, which even has an option for randomized password protection of source distributions.

Window Appearance

The Window Appearance options, shown in Figure 15.8, let you control many details of your VI's window appearance. You can choose between the pre-defined setups of "top-level application window," "dialog," or "default," or you can customize the look.

Figure 15.8. The Window Appearance section of the VI Properties dialog and the Customize Window Appearance dialog floating above it

[View full size image]



Most of the options are self-explanatory; however, a few comments are in order:

- Be careful with options such as disabling the Show Abort Button. A VI with this option can't be hard-stopped, even with the keyboard shortcut, once it's running! (You have to kill LabVIEW to stop it if you didn't provide a programmatic way to abort the VI.) Only use this for VIs you have tested and know will exit properly from the code.
- The Dialog Box option gives the VI an appearance of the standard OS system dialog box, and prevents accessing other LabVIEW windows. (A window that does not allow you to access others is commonly referred to as *modal*.) Be careful with this setting. If you open a modal VI that is not running, but is a subVI of a running VI, then you will be "locked out" and only allowed to interact with the modal VI you just opened. (This is not an uncommon situation. In Activity 15-9, you will create a utility that you can use to rescue yourself from this predicament.)

- The Floating window behavior will cause the window to stay frontmost, but allow you to interact with other windows behind it. For example, the LabVIEW Tools, Controls, and Functions palette windows are *floating* windows.
- Hilite <Enter> BooleanOK, so not all of these options are self-explanatory. This option will highlight (put a black border around) the Boolean control that has been assigned the <Return> or <Enter> key, as discussed in the next section.
- The Window Run-Time Position options allow you to define the window position and size, as well as the monitor where the VI will first appear, when the VI is run.

Window Size

This property allows you to force the VI to be a minimum pixel size. You can also specify settings for letting front panel objects automatically scale if the front panel is re-sized.

Print Options

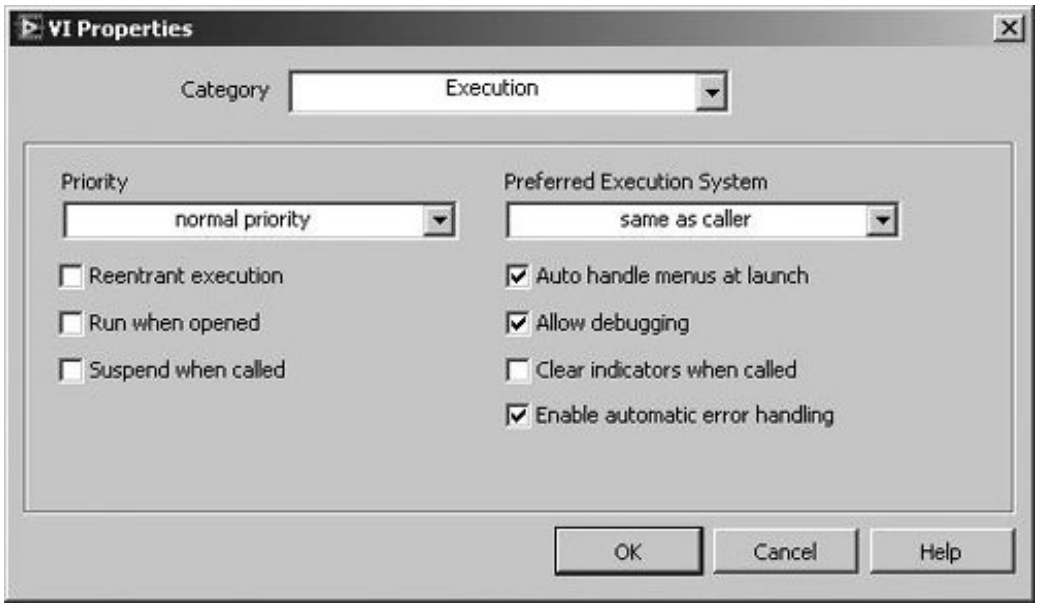
These settings allow you to specify the options like headers and margins for printing.

Execution

The Execution options shown in Figure 15.9 allow you to set some very advanced execution tweaks.

Figure 15.9. Execution section of the VI Properties dialog

[View full size image]



- Priority: Don't mess with this; you shouldn't need it. Trust me on this one. If you really want to know what this is about, read the LabVIEW manuals.
- Reentrant execution deserves a little more explanationwe will discuss it in the next section.
- Run When Opened does just that.
- Suspend When Called is a rarely-used debugging option (and, like the subVI Node Setup options, you should probably avoid it).
- Preferred Execution System: This is LabVIEW's attempt at multi-threading. It allows you to choose one of several threads for your VI to run it. If you don't know what I'm talking about, you won't need it either.
- Auto handle menus at launch causes LabVIEW to automatically handle menu selections in your running VI. If you remove the checkmark from this option, the run-time menu bar is disabled until you use the Get Menu Selection function (found on the Programming>>Dialog & User Interface>>Menu palette) to handle run-time menu selections.
- Allow Debugging, checked by default, gives you access to LabVIEW's debugging tools when the VI is running. You can uncheck it to obtain a slight 12% increase in speed and decrease in memory usage of the VI.
- Clear Indicators when Called causes LabVIEW to reset indicators values to their defaults, when the VI is first runotherwise, they will only change when their values are changed programmatically.
- Enable Automatic Error Handling causes LabVIEW to capture and display an Error Dialog if any VI or function on the block diagram outputs an error that is not wiredwe discussed error handling in Chapter 7, "LabVIEW's Composite Data: Arrays and Clusters."

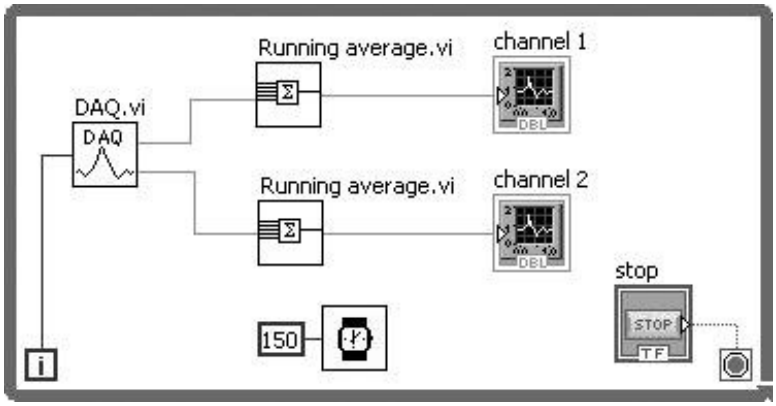
Reentrant Execution

Reentrant execution is an important concept. Normally, when you have more than one instance of a subVI, only one instance can be called at a time (other instances must wait in line until no instance is running and it is their turn to execute). And, normally, all instances share the same data spacethey share the same control and indicator values as well as shift register and other data storage spaces, which is why only one can execute at a time. However, when a subVI is configured for reentrant execution, each instance will have its own data space and can be called independently of (and at the same time as) other instances.

If this sounds confusing, don't worry. Keep reading and study the examples here and you'll begin to see the difference between a reentrant and a non-reentrant VI.

There are two main reasons why you might want to configure a VI for reentrant execution. First, you may not want the non-reentrant behavior where only one instance may be called at a time. If a subVI takes a noticeable amount of time to complete execution and is called in several locations that might execute in parallel, the total execution time of your code might be slowed down by some instances of the subVI having to wait for another instance to finish executing, before they can execute. Second, you may want each subVI instance to have its own data space because they are being used as data storage containers for specific tasks. For example, in the block diagram shown in Figure 15.10, we are using the subVI Running Average on each data channel.

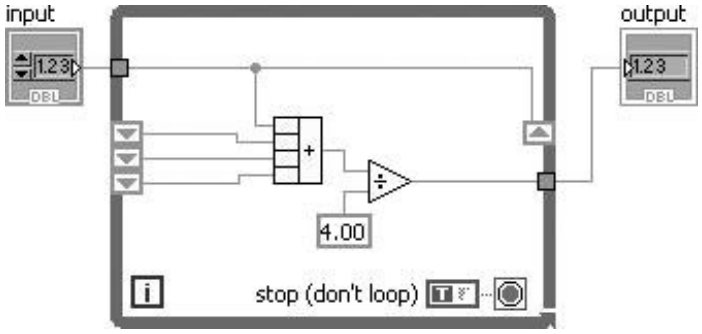
Figure 15.10. Block diagram showing the Running average subVI called in two locations



Running Average uses *uninitialized* shift registers as memory elements. Remember that uninitialized shift registers keep their last value even after the VI is stopped and run again (see Chapter 6, "Controlling Program Execution with Structures," for a discussion of uninitialized shift registers). If this VI were left in the default (non-reentrant) mode, it would give unexpected results because the call to either subVI node would contain the shift register data from the last call to the other node (because the subVIs will normally take turns executing). By choosing the "Reentrant

execution" option, each subVI node is allocated an independent data storage space, just as if they were two completely different subVIs (see Figure 15.11).

Figure 15.11. Running Average.vi block diagram

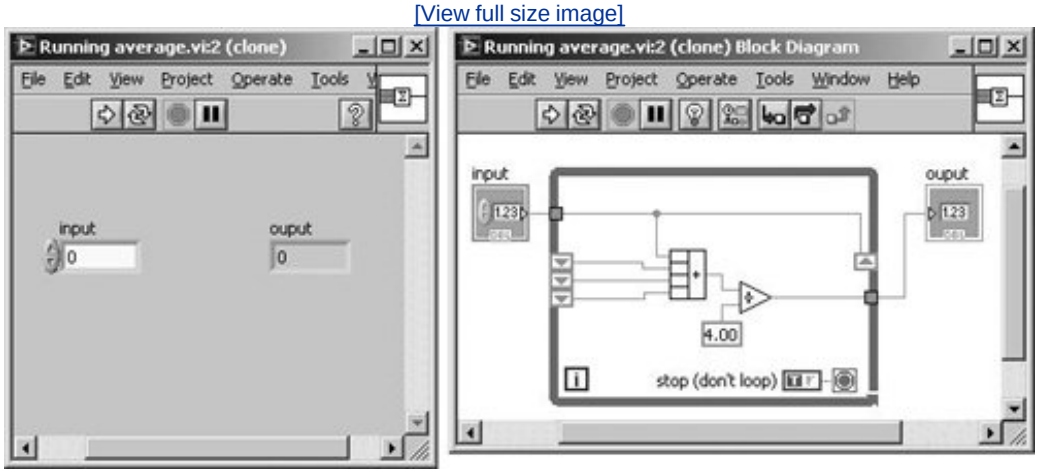


Debugging Reentrant VI Instances ("Clones")

If you open the front panel of a reentrant subVI, as shown in Figure 15.12, the name of the VI in the window title bar is suffixed with a unique instance number and the text "(clone)".

Figure 15.12. Running Average.vi "clone" (reentrant instance)

[View full size image]



Each instance of a reentrant subVI can be opened independently. The reentrant subVI "clones" allow you to interact with the reentrant subVI instances while your application is running. You can debug each clone independently using execution highlighting, single stepping, breakpoints, and wire probes, as well as watching the front panel control and indicator values change while the clone is running.

You will notice that the clones are in *Run Mode* and cannot be edited. However, if you change the clone to *Edit Mode*, by selecting Operate>>Change to Edit Mode from the menu (or using the <ctrl>+M shortcut key combination), the front panel of the reentrant VI (the real VI, not a clone) will appear. The clone(s) will remain open until you make a change to the reentrant VI, at which time they will close, because they are no longer valid (since the VI they were cloned from has changed).

Activity 15-2: Reviewing Recycled Reentrancy

As an interesting activity, first run this VI we just mentioned, Running Average.vi, which is on the CD, to see how it works. Then run the Reentrant VI (also on the CD) first as it is, and then change the "Reentrant Execution" option for the Running Average VI to compare the differences.

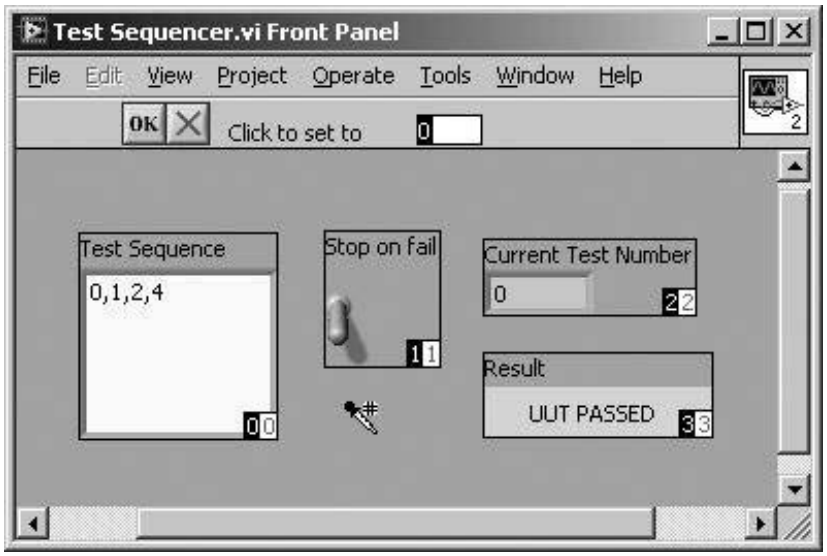
Keyboard Navigation

If you are one of those people who think mice are unfriendly (or your end users think so), there's good newsyou can set up your VI to allow users to "navigate" the controls with the <tab> key (and other keys). Without any special setup, you can always use <tab> to pick the control that will receive input (you can't pick indicators this way, because indicators don't accept inputs). A "selected" controlcalled the *key focus*has a rectangular border enclosing it. Once a control is the key focus, you can use the appropriate keys to enter its value. The following tips may be useful:

- If you directly type the value into the selected control, you must hit the <enter> key when you are done to make your entry valid.
- For numerical and ring controls, you can also use the arrow keys to advance to the desired number. Pressing <shift> with the <up> or <down> arrow key advances the value faster. You can set the minimum increment from the Data Range . . . option in the pop-up menu of the control.
- For Boolean controls, the <return> key toggles the Boolean value.
- Tabbing from control to control normally follows the order in which you created the controls

For VIs with several controls, you may wish to set your own tabbing navigation order; that is, determine once a control is selected, which control will be selected next when the <tab> key is pressed. This sequence is known as *tabbing order* in LabVIEW. To change the panel order, choose Set Tabbing Order . . . from the Edit menu. This works in the same way as the Cluster Order . . . option, discussed in Chapter 7. The front panel in Figure 15.13 shows what it looks like when Tabbing Order is selected.

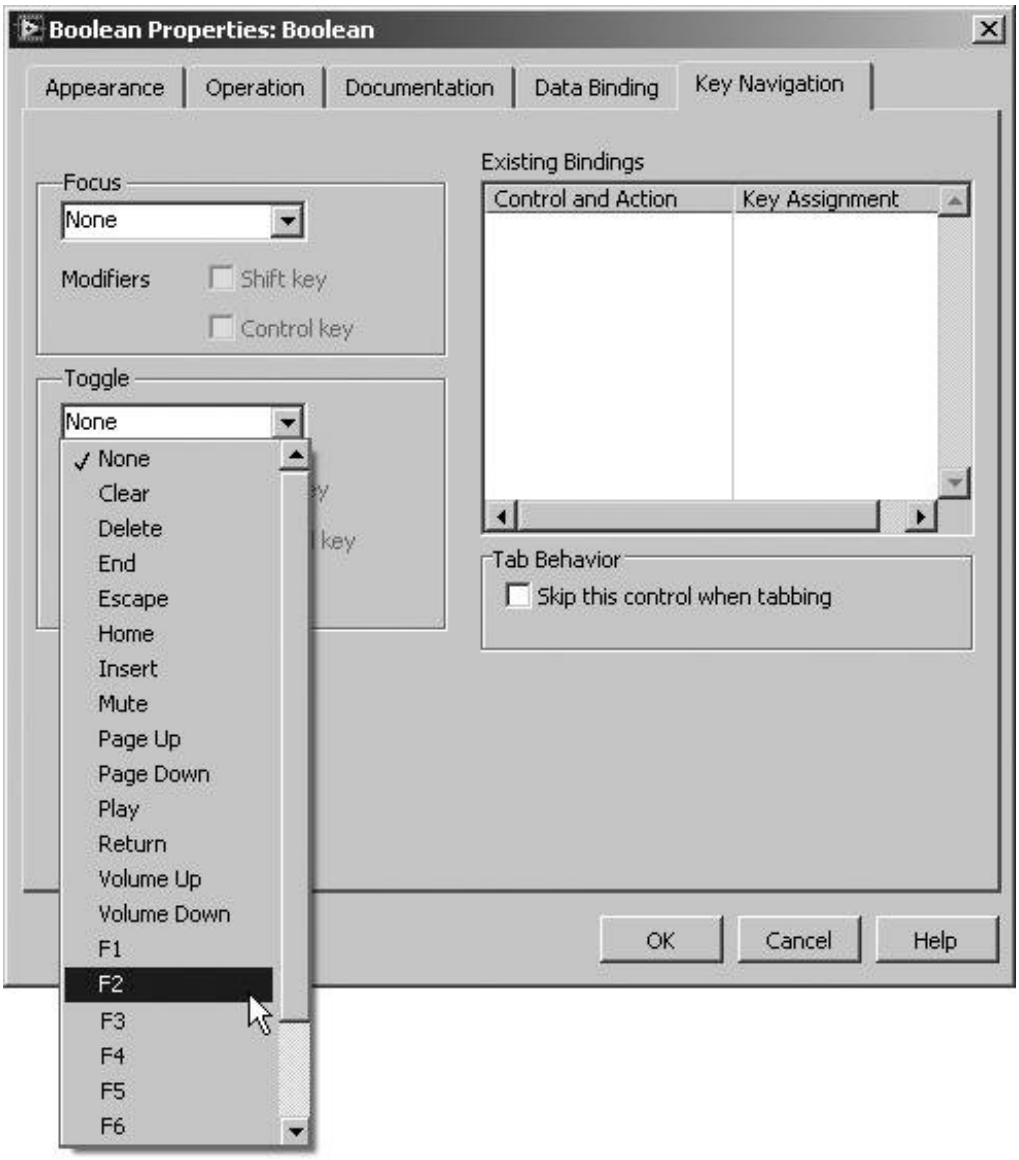
Figure 15.13. Front panel with Set Tabbing Order selected



On your front panel, all controls will be boxed in with two numbers in the lower-right corner of each box. The number in the white background represents the previous panel order; the number in the black background represents the new panel order you are assigning to your diagram. To create the new panel order, click successively on each control in the desired order, or type a sequence number into the text input box on the toolbar that you want to assign to the next control that is clicked, and then click the OK button. To cancel all changes, click the *X* button.

You can also assign certain "special" keys to a control, from the Key Navigation dialog (see Figure 15.14), which is accessible by selecting Advanced>>Key Navigation . . . from a control's pop-up menu.

Figure 15.14. Key Navigation section of the Boolean Properties dialog



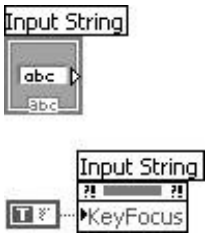
All controls allow you to define a "Focus" key that, when pressed, will select that control. Boolean controls allow you to define a "Toggle" key that, when pressed, will toggle the state of the Boolean (just as if you had pressed it with the mouse). And, numeric controls allow you to define "Increment" and "Decrement" keys that, when pressed, will increment and decrement the numeric's value (respectively).

The Key Navigation dialog also allows you to specify "Skip this control when tabbing," which causes that control to be skipped as the user is pressing the <tab> key to cycle key focus between controls.

You can assign function keys (<F1>, <F2>, etc.) to a control, as well as function keys with modifiers (such as the <shift> or <ctrl> keys). Pressing the selected key will set the key focus on that control without having to "tab" to it. You will find Key Navigation useful if you have many controls but have a few you use more often.

Finally, you can also programmatically set or disable a control's key focus. The Key Focus property of a control is a Boolean that, when true, means the control is "selected" and ready for keyboard input.

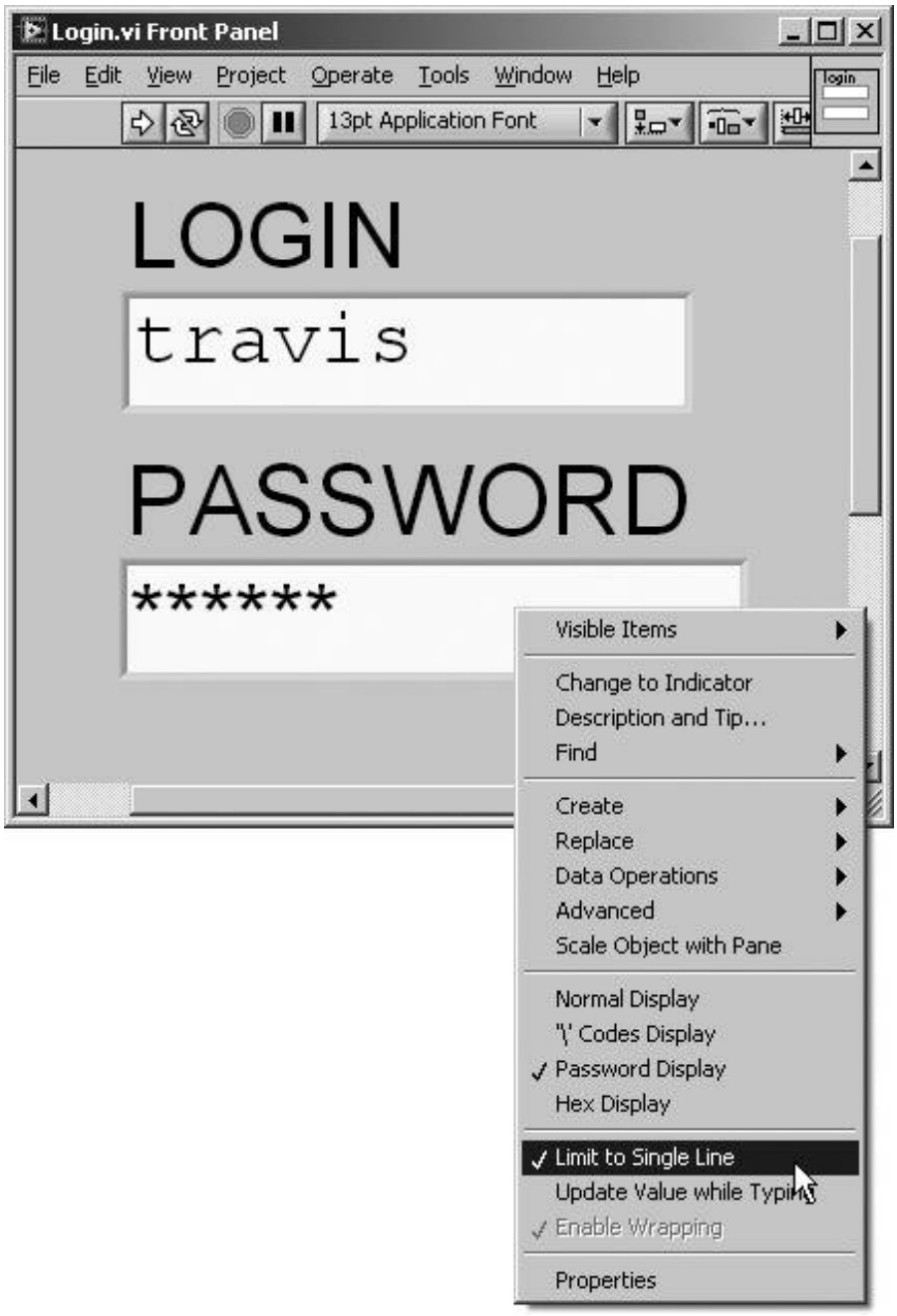
Figure 15.15. Key Focus property of a string control set using a Property Node



Activity 15-3: A Login VI

Build a Login VI with a nice front panel, as shown in Figure 15.16. This VI should make the Key Focus first appear in the LOGIN box. The VI should detect when the user has pressed <return> or <enter> and move the Key Focus to the PASSWORD box. The password characters should not be seen. Finally, the VI should appear in the middle of the screen with the same appearance as a dialog box from the OS. Don't forget to enable the Limit to Single Line options for the string controls on the front panel, as shown in Figure 15.16.

Figure 15.16. Setting the Limit to Single Line option for the **LOGIN** and **PASSWORD** string controls



The following hints will help you complete this activity.

1. *You will need to use the Key Focus control property.*

2. *Pop up on the* `PASSWORD` *control and select the "Password" option for the string.*

3. *Pop up on the* `LOGIN` *and* `PASSWORD` *control and select the "Limit to single line" option.*

4. *How will this VI know when a user has finished the input string to the* `LOGIN`*? When a string control is configured to "Limit to single line," pressing the <return> or <enter> key does not add a new line to the string; rather, it will lose key focus (Key Focus = FALSE). You will need to periodically check whether the control has lost key focus. You can store the key focus value in a shift register and test whether it is currently FALSE AND previously TRUE.*

The solution is found on the CD.

PREV NEXT

The VI Server

The VI Server is a powerful feature in LabVIEW that gives you the capability of programmatically accessing features in LabVIEW like opening and running VIs, changing the color or value of a front panel object, printing the front panel, and so on.

This is quite an advanced topic, so don't feel bad if much of this VI Server introduction confuses you. Almost everyone struggles to understand how to use VI Server functions at first. The good news is that, for simple LabVIEW applications, you might never need it. Once you get warmed up and comfortable with LabVIEW, though, you'll want to play with all the fun and powerful things you can do via the VI Servereventually you'll wonder how you ever programmed in LabVIEW without it.

You don't need to understand or use the VI Server for any of the other material in this book, so feel free to gloss over it if you want to move ahead.

Don't let the name confuse you: "VI Server" is much more than just some type of networking server built into LabVIEW (although it is that as well). The VI Server functionality is really a way of exposing the internal object structure LabVIEW uses for organizing VIs, controls, and a whole lot more.

For example, with VI Server, you can programmatically:

- Load a VI into memory, run it, and then unload the VI, without the need to have it statically linked as a subVI in your block diagram.
- Dynamically run a subVI that gets called at runtime, by only knowing its name and connector pane structure (this is known as calling a VI *by reference*).
- Change properties of a particular VI, such as the size and position of the front panel window, whether it is editable, and so on.
- Make LabVIEW windows move to the front of the screen.
- From the block diagram, call a subVI without waiting for it to finish executing (one of the few places you can get away with not obeying the normal dataflow paradigm!).

- Dynamically change the attributes (properties), such as color and size, of a front panel object.

Because LabVIEW's internal objects (VIs, controls, etc.) are designed using object-oriented programming (OOP) principles, you will see many OOP themes appear in the tools and mechanisms we use with VI Server. For example, the VI Server deals with objects having properties and methods. These objects are organized in a single inheritance hierarchy and are operated on by reference. Although we do not have time to show the relationship between the VI Server features and OOP, your programming will benefit greatly if you take the time to explore OOP on your own and read Appendix D, "LabVIEW Object-Oriented Programming."

In addition to all this cool stuff you can do, there's more: Everything we just mentioned works with *network transparency.* Network transparency means you can do all of the mentioned manipulation of a VI or of LabVIEW itself *on another machine across the network (including the Internet)* in just the same way as if it were on your own machinethe remote LabVIEW can even be running on a different operating system or be a built executable application. This means that, for example, you could have a data acquisition VI running at remote sites while your local analysis VI gets information from the remote machineswithout having to write any special networking and without using complex TCP/IP functions.

Figures 15.17 and 15.18 show you an example of how trivial it is to run a VI remotely. (Don't worry about what the functions are below if you haven't seen them before; they're from the Programming>>Application Control palette, and we'll examine them in detail shortly.)

Figure 15.17. This tells the local machine to run **My Cool VI.vi**.

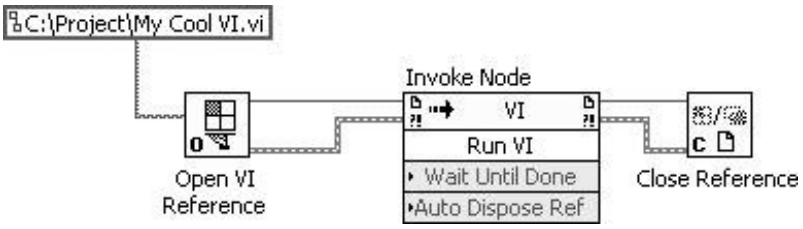
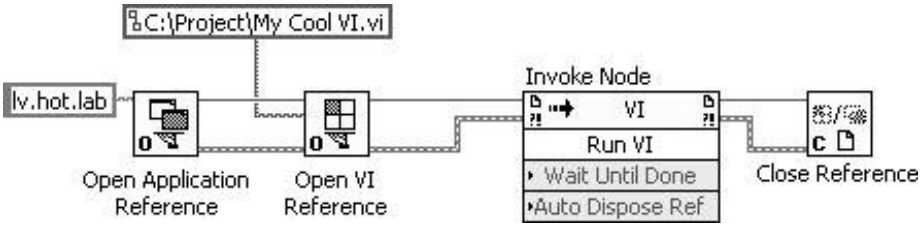
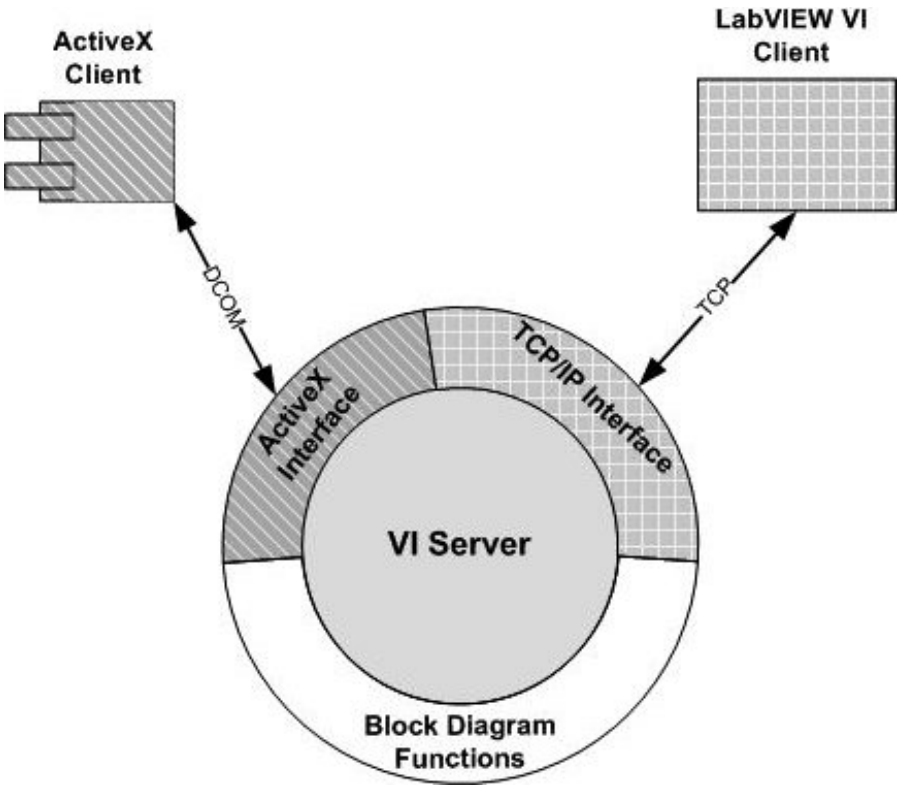


Figure 15.18. This tells the LabVIEW across the Internet at the address **lv.hot.lab** to run **My Cool VI.vi**. The path to the VI refers to the remote drive.



The VI Server exposes its functionality in LabVIEW through block diagram functions like those in Figures 15.17 and 15.18. But it also allows its functionality to be accessed in Windows from external programs through an ActiveX automation client (e.g., a Visual Basic program or macro) and from a remote LabVIEW VI over TCP/IP. The diagram in Figure 15.19 illustrates this architecture. As you can see in the following figure, the VI Server functionality is accessible by block diagram functions, by external ActiveX programs, or by a remote LabVIEW VI over TCP/IP.

Figure 15.19. VI Server communication model



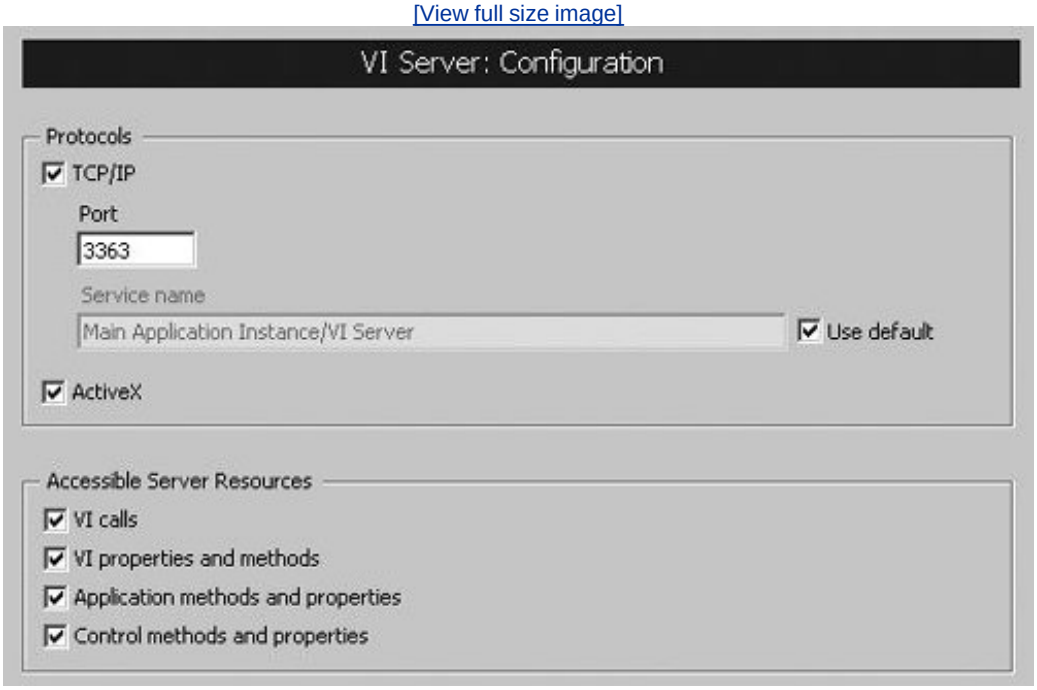
Enabling Remote Access to the VI Server

To enable LabVIEW on remote computers to access the VI Server on your local computer, select

Tools>> Options . . . from the menu to open the LabVIEW Options dialog and then navigate to the VI Server: Configuration category (see Figure 15.20). You can allow access to VI Server via TCP/IP (on all platforms) and via ActiveX (on Windows). Also, pay attention to the various permissions settings for restricting access to certain functionality and VIs, as well as limiting access to certain remote machines and users. There are four options categories that begin with "VI Server:" that are used to configure these settings.

Figure 15.20. LabVIEW VI Server configuration options

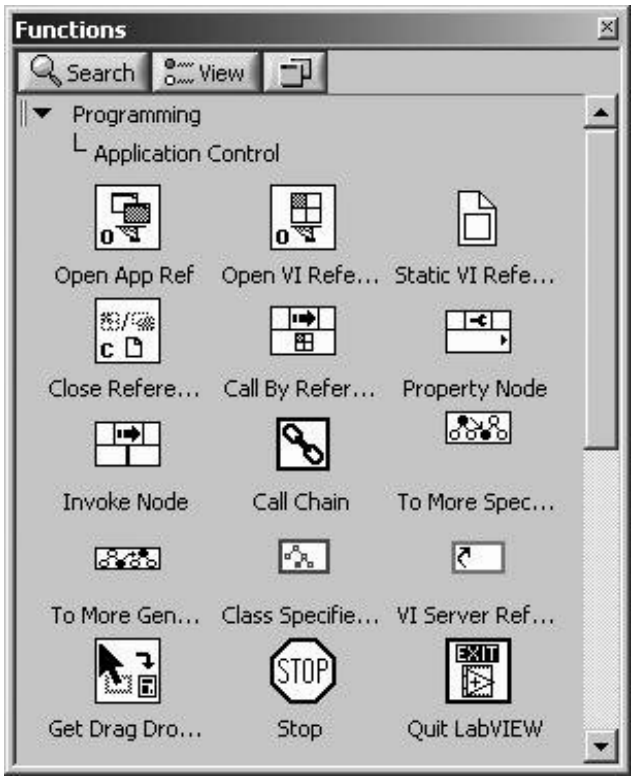
[View full size image]



You always have full access to the VI Server of the local LabVIEW environment in which your VIs are running (regardless of the VI Server settings configured in the Options . . . dialog). If you wish to access the local VI Server, use a local application reference. For example, if you call the Open VI Reference function and leave the `application reference (local)` *input unwired, then the local application reference is used by default.*

The functions for using the VI Server are in the Programming>>Application Control palette (see Figure 15.21).

Figure 15.21. Application Control palette



There are three very important *classes* of "objects" in LabVIEW that the VI Server functions let you manipulate:

- Application class The LabVIEW environment itself.
- VI class A specific VI in memory or on a disk.
- Control class A front panel control or indicator.

We will discuss each of these in the coming sections.

Properties and Methods, "By Reference"

The term "by reference" means that we are passing around a reference (also called a "refnum," "pointer," or "handle") to objects that we wish to operate on. We learned how to perform operations "by reference" when we studied the file I/O functions in Chapter 14, "Advanced LabVIEW Data

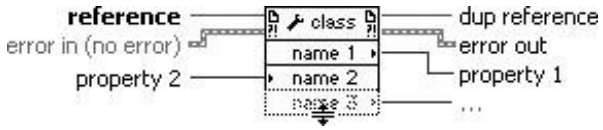
Concepts." There, we used the *file refnum* to tell the file I/O functions which file we wished to perform operations on.

VI Server is another example of performing operations by reference. We will open application, VI, and control references (similar to how we opened file references) and then read and write properties and invoke methods by reference (similar to how we called the file I/O functions to operate on a file by reference).

For operating on VI Server objects, we will primarily use the Property Node and Invoke Node (Programming>>Application Control palette).

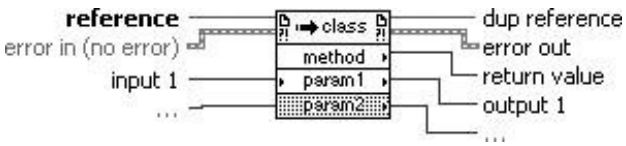
The Property Node sets (writes) or gets (reads) property information for the object whose reference is passed into the reference input (see Figure 15.22). You can configure a Property Node to access multiple properties; each property can be configured as read or write. Pop up on a property terminal and use the Change To Read, Change To Write, Change All To Read, or Change All To Write options accordingly. You can resize a Property Node to show more or fewer terminals. Each terminal can be used to read or write one property.

Figure 15.22. Property Node



The Invoke Node invokes a method or action on the object whose reference is passed into the reference input (see Figure 15.23). Once you select the method, the associated parameters will appear below the method name. You can set and get the parameter values. Parameters with a white background are required inputs and parameters with a gray background are recommended inputs. You can only configure an Invoke Node to call a single method. The number of terminals is defined by the number of arguments and return valuesit cannot be resized.

Figure 15.23. Invoke Node



If the reference input is wired, the Property Node or Invoke Node will automatically adapt to the

class of the object that is referenced. If the reference input is unwired, then you can select the class of object from the Select Class pop-up submenu. If you select the VI or Application class, you do not need to wire the reference inputLabVIEW assumes that you are referring to the calling VI or the local application (respectively).

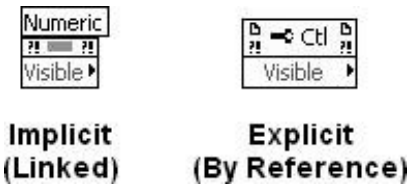
Déjà Vu: Property Nodes and Invoke Nodes

In Chapter 13, "Advanced LabVIEW Structures and Functions," you also learned about Property Nodes and Invoke Nodes, but those were *linked* to controls. You create Property Nodes and Invoke Nodes by selecting a property or method from the Create>>Property Node or Create>>Invoke Node submenus (respectively) of a control's pop-up menu. This created a Property Node or Invoke Node on the block diagram that did not have a reference input or dup reference output terminal. It did not need those terminals, because the reference (or link) to the control was implicitly defined.

If you have explored the pop-up menu of a Property Node or Invoke Node that is linked to a front panel control, you may have noticed the option Disconnect From Control. This creates an *unlinked* property or Invoke Node that looks identical to the one on the Programming>>Application Control palette. (And, yes, conversely you can pop up on an *unlinked* property or Invoke Node and link it to a front panel control.)

You can see the difference between an implicit (linked) and explicit (unlinked) Property Node in Figure 15.24.

Figure 15.24. Implicit (linked) and explicit (by reference) Property Nodes



That is all we will say for the moment about control references. You'll learn more about them soon. First, you will learn about application references and VI references, in order to better understand how control references fit into the picture.

Application References

An application reference is a reference to an instance of the Application class in LabVIEW. What this means is that an application reference gives you access to read and write properties and to invoke methods on the LabVIEW application itself (as opposed to a single VI).

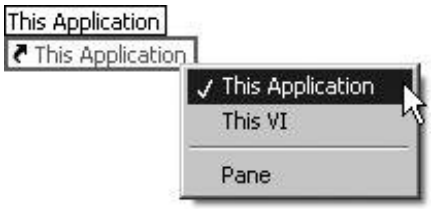
You can also open an application reference to a LabVIEW standalone executable application (both local and remote application references). But, you must configure the built application's INI file settings. The configuration file key-value pairs used for VI Server (and many other) settings in a built application are the same as those used by the LabVIEW development environment. Note that the INI file section must be the name of your built application (minus the file extension).

For example, suppose your program needs to know if your user's monitor resolution is the right size. With an application reference, you can read what monitor resolution LabVIEW is running by reading the "Display.AllMonitors" property. Or let's say your program launches an external application, such as Microsoft Excel, but you want to make sure the VI window comes back to the front of the screen. You can invoke the "Bring To Front" method, which tells LabVIEW to move its windows to the front of other windows.

Before you can read or write application properties using a Property Node, or call application methods using an Invoke Node, you will need to create (obtain) the application reference. Usually the application reference refers to the local LabVIEW installation, but it could refer to a LabVIEW installation on a remote machine. Obtaining a local application reference is easy and there are a few ways to do so.

One way is to use a VI Server Reference (Programming>>Application Control palette) and configure it for This Application from the menu by mouse-clicking on it with the Operating tool (see Figure 15.25).

Figure 15.25. VI Server Reference

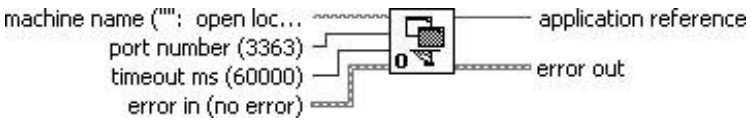


Another way is to use an Application Property Node (Programming>>Application Control palette) and leave its reference input unwired, as shown in Figure 15.26. Doing so will cause LabVIEW to assume that you are referring to the local application and the `dup reference` output will be a local application reference.

Figure 15.26. Application Property Node

The last way to obtain a local application reference is to call the Open Application Reference function (see Figure 15.27) and leave the machine name unwired (causing the function to assume the default: local application).

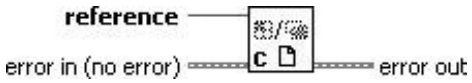
Figure 15.27. Open Application Reference



You can also use the Open Application Reference function to obtain a *remote* application reference (a reference to the VI Server on a remote LabVIEW). To obtain a remote application reference, specify a machine name (host name or IP address). This will establish a TCP connection with the remote VI Server on the specified port number.

When you are done using a remote application reference that was created by Open Application Reference, you must close it by passing the reference to the Close Reference function (Programming>>Application Control palette), shown in Figure 15.28.

Figure 15.28. Close Reference



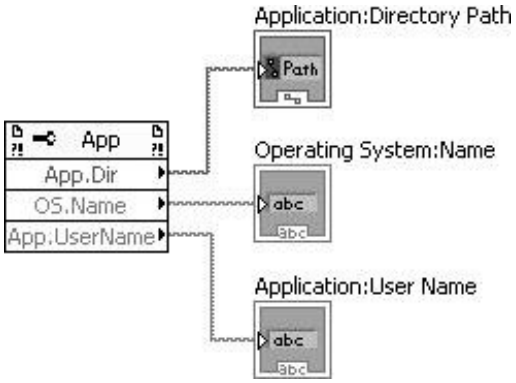
Close Reference is also used to close any refnum associated with an open VI, VI object, an open application instance, or an ActiveX or .NET object. You will see this function often when using VI Server.

Activity 15-4: Using the Application Class Properties

In this activity, you will use the VI Server to find out what operating system you are running, where LabVIEW is installed, and the username of who is logged on.

1. With a new blank VI, create the following block diagram by making an Application Property Node (from the Programming>>Application Control palette). You will be programmatically reading the directory LabVIEW is in, the OS name, and the LabVIEW user name (see Figure 15.29).

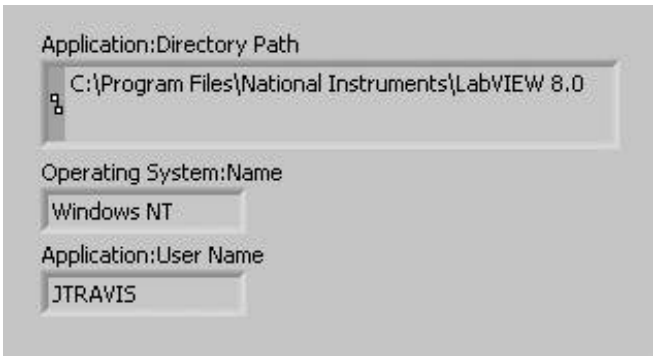
Figure 15.29. Block diagram of the VI you will create during this activity



When you are referencing the local Application class, you can omit the Open Application Reference function that normally would precede the Invoke Node function.

2. On the front panel, run the VI and observe the results (see Figure 15.30).

Figure 15.30. Front panel of the VI you will create during this activity, showing the results of running it



3. Save your VI as AppClassDemo.vi.

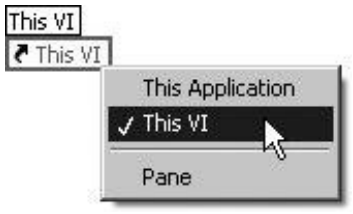
Now let's look at how you can use the VI Server to create some "magic": Allow one VI to change the value of another VI's front panel control, without wiring the two together.

VI References

A VI Reference refers to an instance of the VI Class. This means you can read/write properties or invoke methods for a specific VI. You can obtain VI References in a similar fashion to application references.

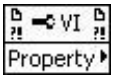
One way is to use a VI Server Reference (Programming>>Application Control palette) and configure it for This VI from the menu by mouse-clicking on it with the Operating tool (see Figure 15.31).

Figure 15.31. Configuring a VI Server Reference to return a reference to This VI



Another way is to use a VI Property Node and leave its reference input unwired, as shown in Figure 15.32. Doing so will cause LabVIEW to assume that you are referring to the calling VI (the VI on whose block diagram the Property Node is placed) and the `dup reference` output will be a reference to the calling VI.

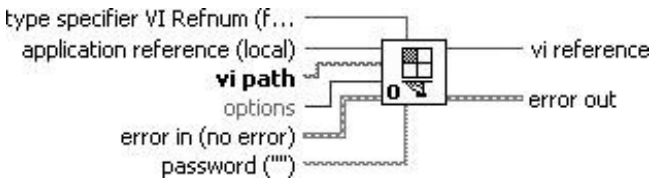
Figure 15.32. VI Property Node



To create a VI Property Node, place a Property Node (Programming>>Application Control palette) onto the block diagram of your VI and then choose Select Class>>VI Server>>VI>>VI from the pop-up menu of the Property Node.

The final way to obtain a VI Reference is to call the Open VI Reference function (see Figure 15.33). This function returns a reference to a VI specified by the `vi path` input. This input can be either a string or a path data type and can be either the VI Name (if it is already in memory) or the full path to the VI if it is not in memory.

Figure 15.33. Open VI Reference



When using Open VI Reference with a remote application reference, the `vi path` input must be specified relative to the remote machine's file system. The remote machine cannot (necessarily) access files in the local machine's file system.

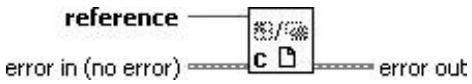
The Open VI Reference function can be used to open references to custom control (`.ctl` files) or global variables, both of which are treated as VIsactually, they are special types of

VIswithin the VI Server class hierarchy.

The `options` and `type specifier VI Refnum` inputs of the Open VI Reference function affect how you will be able to use the VI reference. We will describe the implications of these inputs, in context, as they relate to the features described in the coming sections.

Any reference created using the Open VI Reference function must be closed using the Close Reference function (see Figure 15.34). Failing to do so can result in your application gradually consuming more and more memory (and possibly degrade system performance).

Figure 15.34. Close Reference



Activity 15-5: Using the VI Class Methods and Properties

In this activity, you will manipulate one VI and toggle one of its controls through the VI Server interface from a "master" VI.

1. Create the simple Chart.vi, as shown in Figures 15.35 and 15.36.

Figure 15.35. Front panel of the VI you will create during this activity

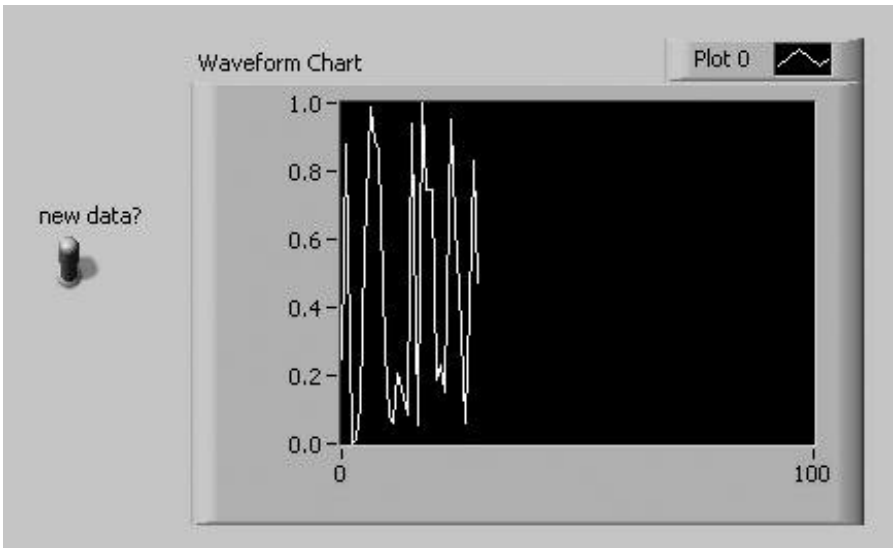
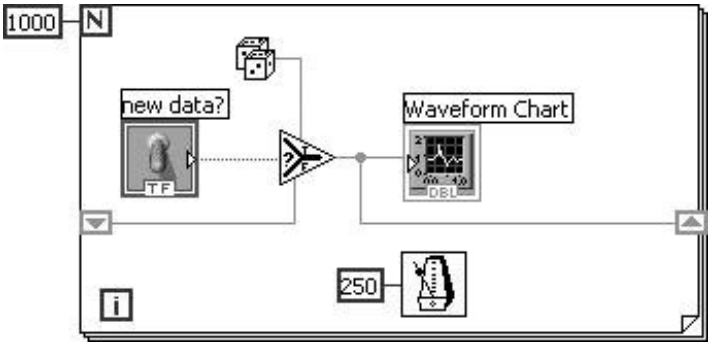


Figure 15.36. Block diagram of the VI you will create during this activity



2. Now open the VI on the CD in `EVERYONE\CH15`, Master VI.vi and run it. Make sure that Chart.vi is still open and you are *not running it*.

3. Run Master VI.vi. It will do the following:

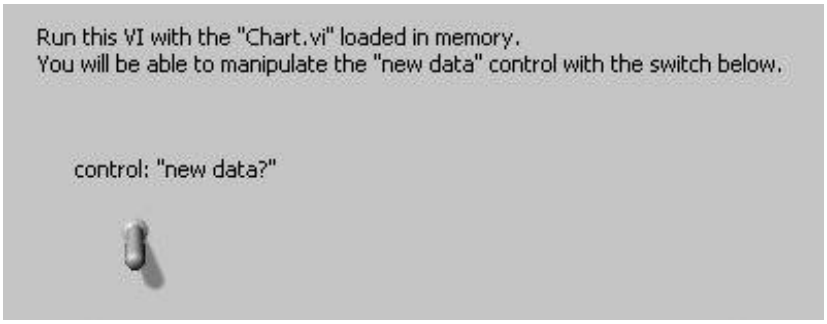
    i. Open the front panel of Chart.vi.

    ii. Center the front panel on the screen.

    iii. Run the Chart.vi.

    iv. Allows the "new data?" control to be changed while running (see Figure 15.37).

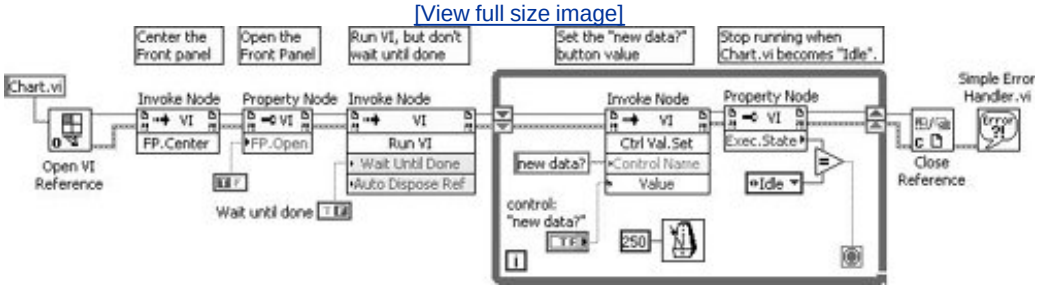
Figure 15.37. Master VI.vi front panel



Study the block diagram to gain insight into how it accomplishes this (see Figure 15.38). Note the use of floating label comments to clearly describe the function of each portion of the code. (This is not overkill; this is good coding practice. Aren't you glad it's there for you to read?)

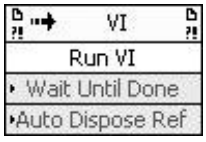
Figure 15.38. Master VI.vi block diagram

[View full size image]



This example uses an Invoke Node to execute the VI's Run VI method, shown in Figure 15.39.

Figure 15.39. Invoke Node configured to execute a VI's Run VI method



Run Button

The Run VI method produces the same result as pressing the VI's Run button on the toolbar of its front panel or block diagram. The Run VI method of executing a VI is different from calling a subVI

because we do not pass data into the VI, or receive data out of the VI, through its connector pane. Also, unlike calling a subVI, we can choose whether we want the Run VI method to wait until the VI finishes executing by passing the desired value to the `Wait Until Done` input argument.

When a VI is run this way, it uses the current values of all front panel controls for execution rather than using data passed in through its connector pane. This method also ignores the Execution:Show Front Panel On Call and Execution:Close After Call properties of a VI. And, you cannot use this method to run a VI that is already reserved for execution by another VI (used as a subVI in the hierarchy of a running VI).

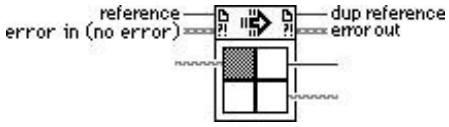
If you use the Open VI Reference function and wire the `type specifier VI Refnum` input, you cannot use the reference returned by the function with the Run VI method. Instead, you must use the Call By Reference Node. We will learn about the Call By Reference Node next.

If you want to use the Run VI method to run a reentrant VI, set the option's parameter to 0x08 in the Open VI Reference function to prepare the VI for reentrant run.

Dynamic SubVIs: The Call By Reference Node

When you want to dynamically call a VI (by reference) as a subVI, you will need to use the Call By Reference Node (Programming>>Application Control palette). This is very similar to a subVI node, but it has reference and error terminals at the top of it (see Figure 15.40).

Figure 15.40. Call By Reference Node



The connector pane of the Call By Reference Node adapts to the connector pane of the VI reference wired into its `reference` input terminal. This connector pane association is defined at the time the VI reference is opened using the `type specifier VI refnum` input of the Open VI Reference function.

However, the Call By Reference Node can also be "statically linked" to a specific VI, in which case it does not require a VI reference to be wired into it. To statically link a VI to a Call By Reference Node, pop up on it and select Call Setup . . . (see Figure 15.41) to open the VI Call Configuration dialog (see Figure 15.42).

Figure 15.41. Selecting Call Setup . . . from the Call By Reference Node's pop-up menu to open its VI Call Configuration dialog

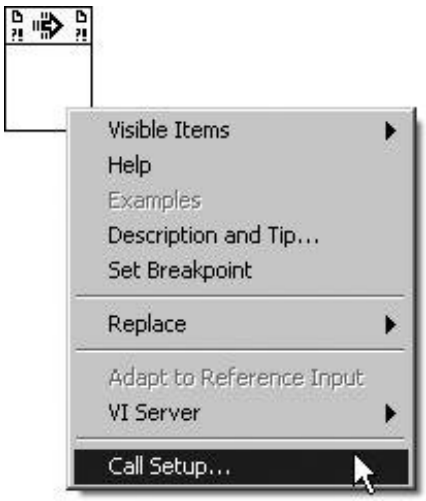
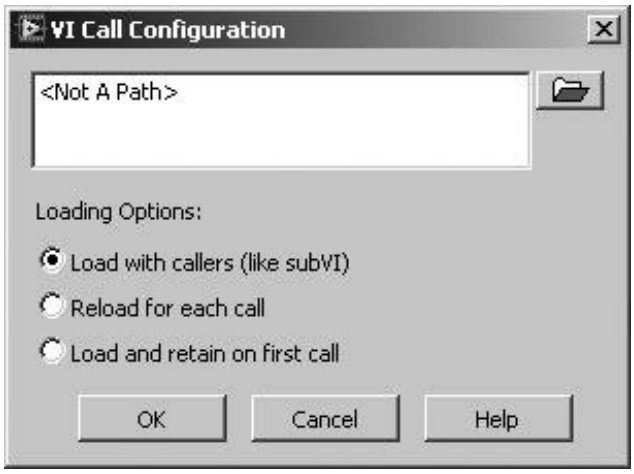


Figure 15.42. VI Call Configuration dialog



If you use the Call Setup . . . option and statically link to a VI, it is very similar to calling a subVI. If you select the "Reload for each call" option, though, the calling VI will load and unload the VI you are calling from memory (whereas a subVI always remains in memory until its calling VI is closed).

This may seem a bit confusing, so let's clarify it with a hands-on example.

Activity 15-6: Calling a VI by Reference

In this activity, you will use the Call By Reference Node to call a subVI by reference.

1. Create Running average by reference.vi, as shown in Figures 15.43 and 15.44.

Figure 15.43. Front panel of the VI you will create during this activity

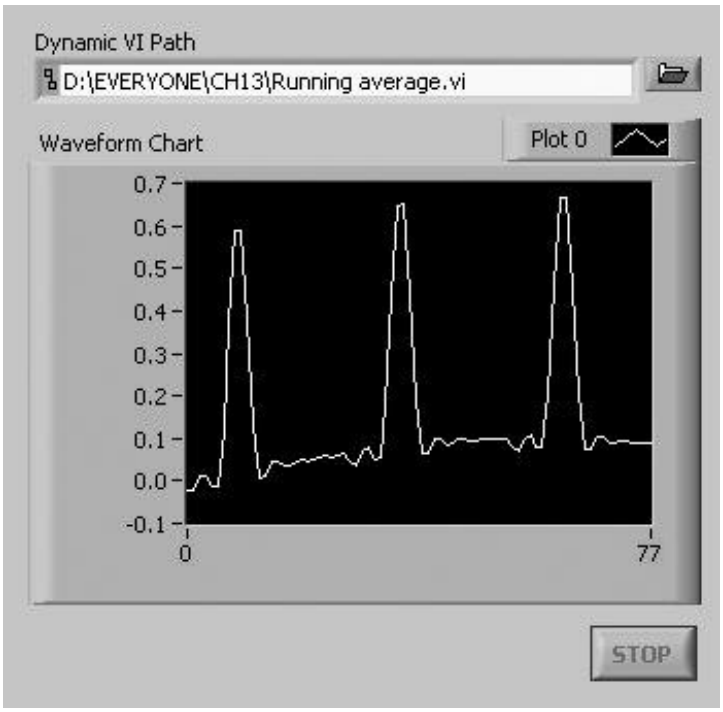
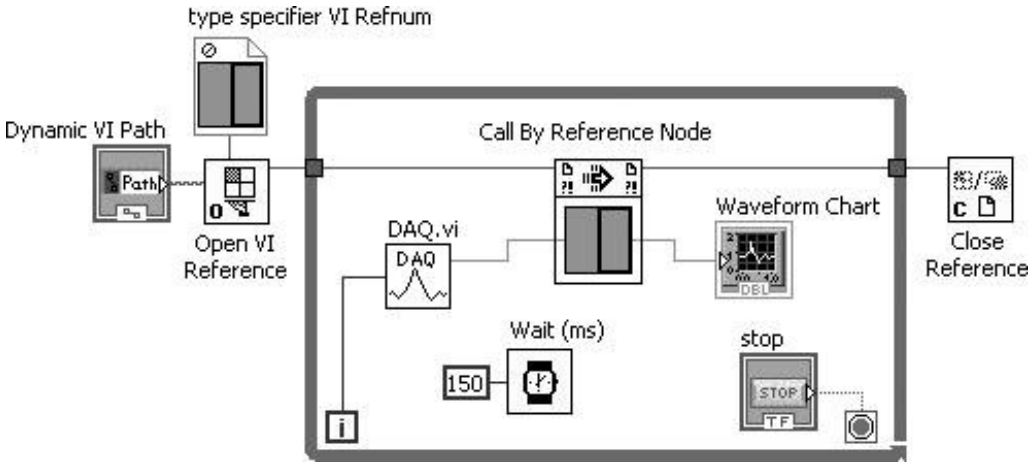


Figure 15.44. Block diagram of the VI you will create during this activity



2. In order to define the connector pane of the Call By Reference Node, we must configure the Type Specifier VI Refnum to define the VI reference wire type. To create the Type Specifier VI Refnum, pop up on the `type specifier VI Refnum` terminal of the Open VI Reference function and select Create>>Constant. Note that you must make sure that your mouse is *directly* over the `type specifier VI Refnum` terminal in order to do this step.

OpenVI Reference

3. The Type Specifier VI Refnum constant must now be associated with the connector pane of Running average.vi. To do this, pop up on the Type Specifier VI Refnum and choose Select VI Server Class>>Browse . . . . A file dialog will open. Browse to the location of Running average.vi (on the CD in `EVERYONE\CH13`), select Running average.vi, and press "OK." The connector pane of Running average.vi will appear in the Type Specifier VI Refnum constant. The Call By Reference Node will also assume the connector pane, if wired to the `vi reference` output of Open VI Reference.

Type Specifier VI Refnum

4. Set the value of the `Dynamic VI Path` control to the path of Running average.vi, located in the `EVERYONE\CH13` folder of the CD.

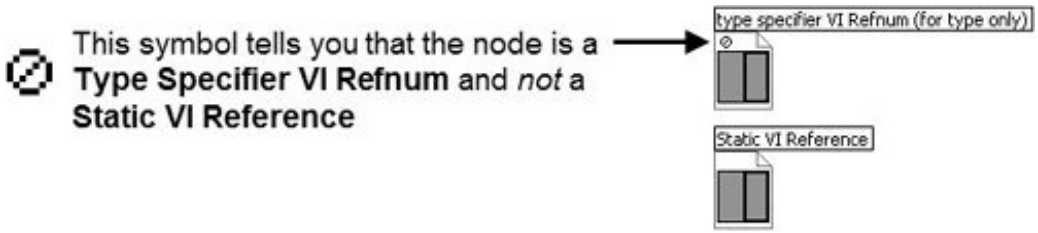
5. Run Running average by reference.vi. You will see data appear on the waveform chart.

In the last step, you associated the Type Specifier VI Refnum constant with the connector pane of Running average.vi. This association does not create any "linkage" or "reference" to Running average.vi. It is simply a way for you to tell the Type Specifier VI Refnum constant the connector pane pattern of the VI that you will later call by reference (which may or may not be the same VI).

Figure 15.45 shows a Type Specifier VI Refnum and a Static VI Reference (which is discussed immediately following this activity), side-by-side, both of which have been "associated" with the same VI (Running average.vi). Make note of the symbol in the upper-left corner that distinctively identifies the Type Specifier VI Refnum. This symbol implies that it does not actually contain a referenceit is "empty"unlike the Static VI Reference, which does actually refer to a VI and contain a valid reference.

Figure 15.45. Type Specifier VI Refnum (top) and a Static VI Reference (bottom)

[View full size image]



You are now calling Running average.vi dynamically! It does not get loaded into memory until you run Running average by reference.vi, and it gets unloaded when Running average by reference.vi stops running. We can think of Running average.vi as a *plug-in* component to our applicationvery cool!

Static VI References

In some instances it is desirable to open a reference to a VI staticallyfor example, when you know, at the time the application is being created, precisely which VIs you wish to open references to.

Here is how it works. Place a Static VI Reference (Programming>>Application Control palette) onto the block diagram. When the Static VI Reference is first placed onto the block diagram, it appears with a question mark icon (see Figure 15.46), signifying that it does not yet reference any VI.

Figure 15.46. Static VI Reference (Unconfigured)

To configure the Static VI Reference to reference a VI, choose Browse for Path . . . from its pop-up menu and use the resulting file dialog to select the VI that is to be referenced (see Figure 15.47). Once a VI is referenced by this node, its icon will replace the question mark icon and you can open its front panel by double-clicking the VI's icon.

Figure 15.47. Static VI Reference (Referencing a VI)

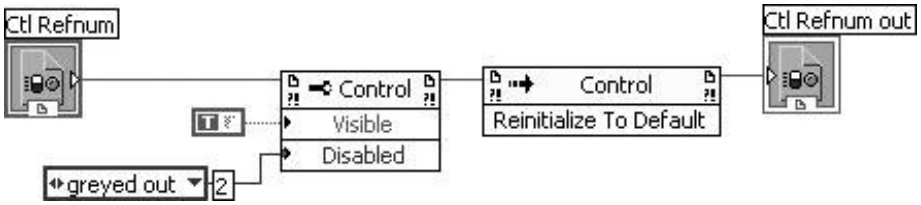
The Static VI Reference (Programming>>Application Control palette) acts similar to a subVI, in that the statically referenced VI appears in the hierarchy of its caller. However, when the Static VI Reference is called, it does not *call* the VI being referenced; it merely outputs a reference to the statically referenced VI.

By default, the Static VI Reference outputs a generic VI reference, but you can change it to be strictly typed (so that the reference contains connector pane information and may be used with the Call By Reference Node) by right-clicking the function and selecting Strictly Typed VI Reference from the shortcut menu. When configured as strictly typed, a Static VI Reference has a red star in its upper-left corner. You should note that when you create a strictly typed VI reference, you cannot use it with the Run VI method.

Control References

A Control Reference is similar to the Application Reference and VI Reference. It refers to a specific front panel object (for example, a numeric indicator). We can pass a control reference to a Property Node in order to get and set properties of the control. Or we can pass the control reference to an Invoke Node in order to call a method the control. Figure 15.48 shows a control reference being used to set properties and invoke a method of a control.

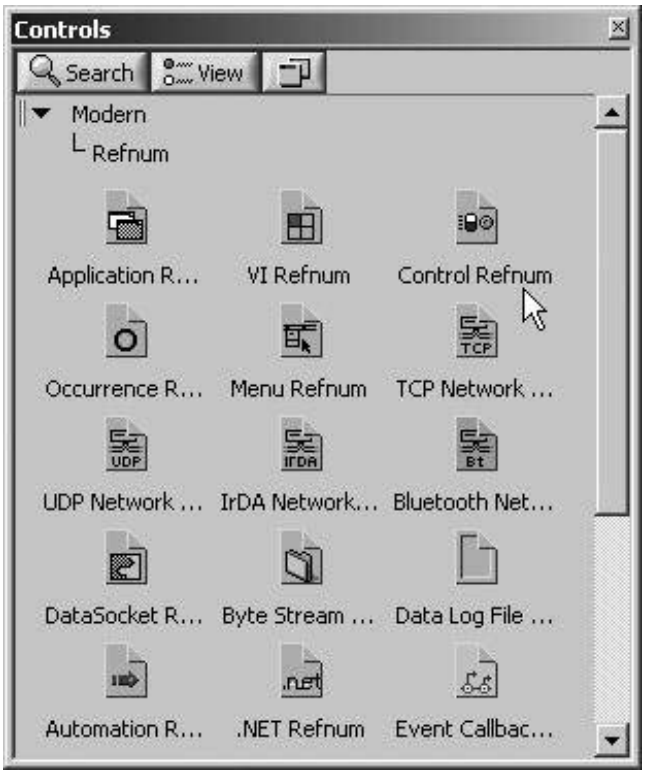
Figure 15.48. Using a control reference to set properties and invoke a method of a control



Control references can be passed into and out of subVIs using a Control Refnum (see `Ctl Refnum` and `Ctl Refnum out` in Figure 15.48).

The Control Refnum can be found on the Modern>>Refnum palette, shown in Figure 15.49.

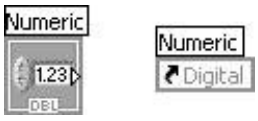
Figure 15.49. Control Refnum, found on the Modern>>Refnum palette



VI Server References to Controls

The simplest way to create a control reference is to pop up on a control or indicator and select Create>>Reference, which will create a VI Server Reference on the block diagram that is *linked* to the control (see Figure 15.50). When a VI Server Reference node is linked to a control, its label text will always be the same as the control to which it is linked.

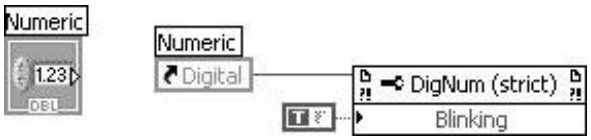
Figure 15.50. Numeric control with a linked VI Server Reference node next to it



The control reference may be wired to Property Nodes or Invoke Nodes, as shown in Figure 15.51.

Figure 15.51. VI Server Reference wired to a Property Node used to set

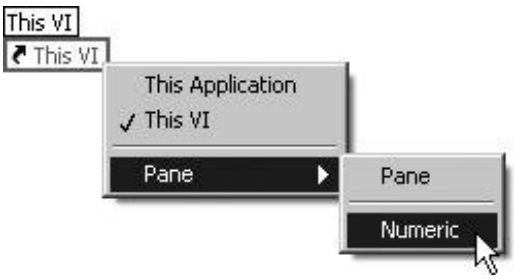
properties of the numeric control



You can also create a VI Server Reference that is linked to a control by dropping a VI Server Reference node (Programming>>Application Control palette) onto the block diagram.

Once you have placed the VI Server Reference onto the block diagram, link it to the desired front panel control by mouse-clicking on it with the Operating tool and selecting the desired control from the list of front panel controls on the Pane submenu, as shown in Figure 15.52. Or you can pop up on the VI Server Reference and select the desired front panel control from the Link To>>Pane submenu.

Figure 15.52. Linking a VI Server Reference to a control on the front panel "pane" from its pop up menu

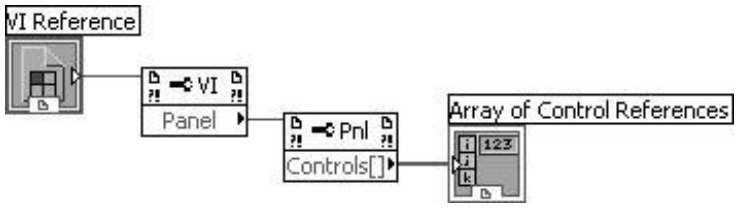


It is important not to confuse the VI Server Reference ( ) and the Class Specifier Constant ( ), which are on the same palette. The VI Server Reference is linked to a specific LabVIEW object (an Application Instance, a VI, or a front panel control or indicator) and outputs a reference that points to a specific LabVIEW object. The Class Specifier Constant, on the other hand, is not linked to any LabVIEW objectits value is null, and it is useful only for its type information. This distinction will be very important when we discuss the VI Server class hierarchy and how to convert control references to more specific or more generic types.

Obtaining References to All Controls on a Front Panel

Another way to obtain control references is by reading the value of the Controls[ ] property of a VI's Panel (the value of a VI's Panel property is a reference to its front panel). This method, illustrated in Figure 15.53, will return an array of control references: one for each control and indicator on a VI's front panel.

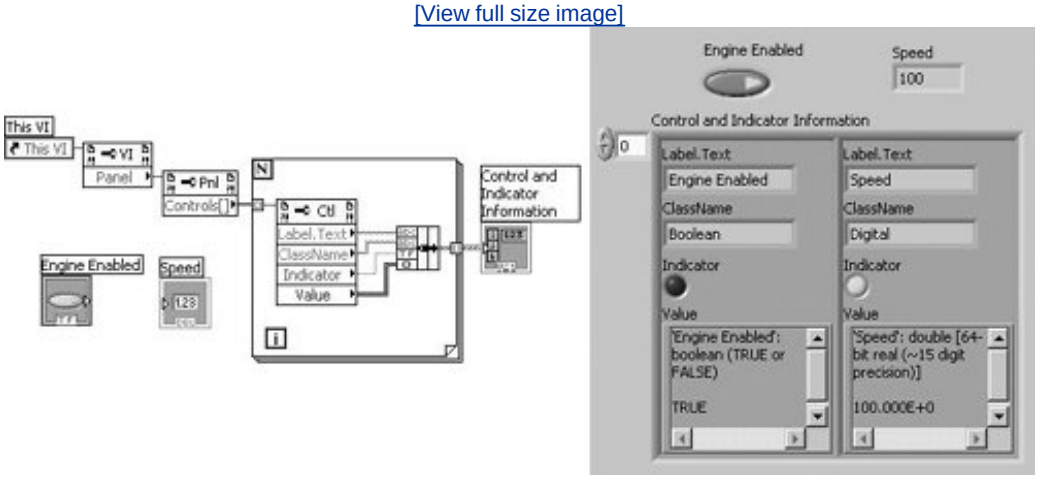
Figure 15.53. Programmatically obtaining references to all controls on a VI's front panel



Using this method, you can obtain an array of control referencesbut how in the world do you know which reference belongs to which control or indicator? Well, one way is to analyze each control reference, one by one, looking at the name, type, and other properties of the control or indicator that the reference points to. Figure 15.54 shows how we can read the Label.Text (Control/Indicator Name), ClassName (Type of Control), Indicator (Control or Indicator), and Value properties of each control on the front panel.

Figure 15.54. Programmatically building an array of properties for all controls on a VI's front panel

[View full size image]



Another, sometimes easier, way you can determine which control references refer to which front panel objects in the Controls[] array is to examine the Tabbing Order of the front panel (the Tabbing Order is accessible from the Edit>>Set Tabbing Order menu option on the front panel; refer to the "Keyboard Navigation" section of this chapter). The Controls [] array returns references to the VI's front panel controls in the same order as they appear in the Tabbing Order.

Using the Tabbing Order to find the index of a control in the Controls[] array may be a useful trick, but it is bad style! The linkage between tabbing order and the control's position in the Controls[] array is quite obfuscated and will give you a headache when you have to figure out how your code works one month from now. If you need to create a reference to a specific control on the front panel, then use the Create>>Reference option from the pop-up menu.

Note that in the example illustrated in Figure 15.54, the Value property is a variantnot a Boolean or a numeric data type. The reason for this is that the control references that come out of the Controls[] property are generically typed. We do not necessarily know, until the moment that we run this code, what specific type of controls we have on the front panel (we will learn about the class hierarchy of control types next). So, we need a generic data type for passing the control's value that can accept *any possible data type*. This is exactly why variants are important and useful. Refer back to Chapter 12, "Instrument Control in LabVIEW," if you want to review the topic of variants.

Activity 15-7: Using VI Server References to Disable Controls While Busy

In this activity, you will create a VI that grays out (disables) a group of controls while data is being generated and sent to a waveform chart.

1. Create Disable Controls While Busy.vi, as shown in Figures 15.55 and 15.56.

Figure 15.55. Front panel of the VI you will create during this activity

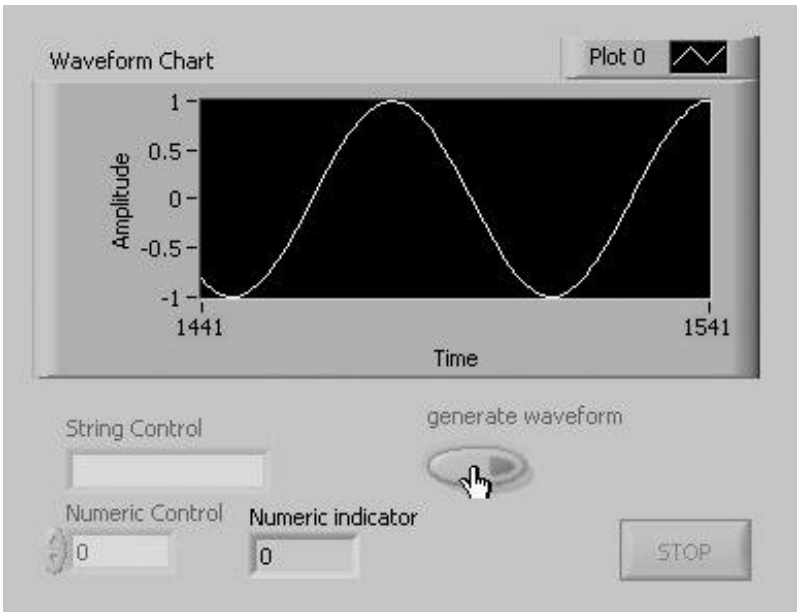
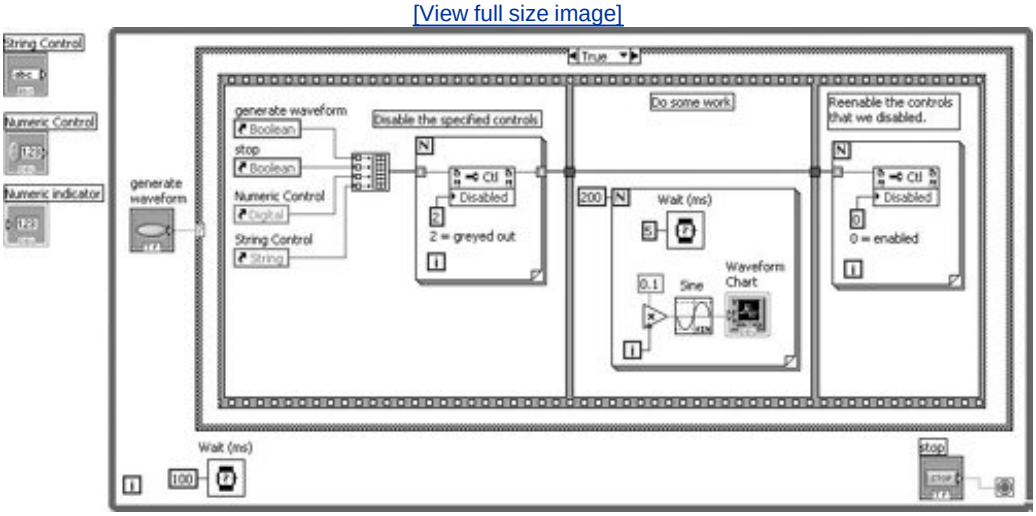


Figure 15.56. Block diagram of the VI you will create during this activity

[View full size image]



2. Create a VI Server Reference for each control you wish to disable by selecting Create>>Reference from its pop-up menu. Use the Build Array node to construct an array of these control references.

VI Server Reference

3. Inside a For Loop (and before doing the work of sending data to a waveform chart), disable each of the controls by setting its Disabled property to 2 (grayed out).

4. After doing the work, re-enable each by setting its Disabled property to 0 (enabled).

Run your VI and press the `generate waveform` button. Watch the controls become grayed out while data is being generated and become re-enabled after the data generation is complete!

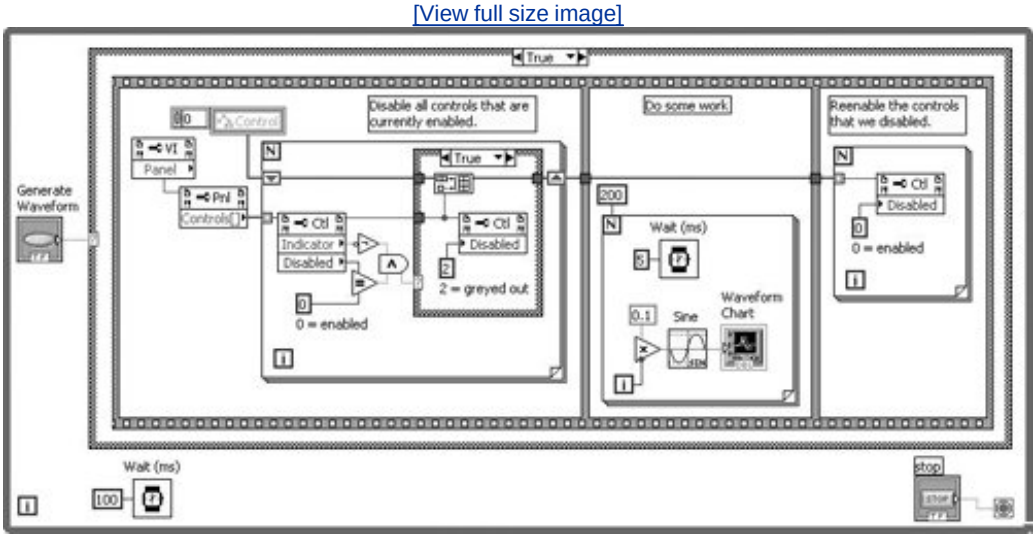
Activity 15-8: Using the Panel.Controls[] Property to Disable Controls While Busy

In this activity, you will modify Disable Controls While Busy.vi, which you just created in the last activity, so that it grays out (disables) *all* of the front panel controls while data is being generated and sent to a waveform chart (not just a group of controls that you specify using VI Server Reference nodes).

1. Open Disable Controls While Busy.vi and save a copy as Disable All Controls While Busy.vi. Modify its block diagram so that the code inside the While Loop looks like Figure 15.57 (the controls outside the While Loop are not shown in the illustration).

Figure 15.57. Block diagram of the VI you will create during this activity

[View full size image]



2. Obtain the VI's Panel reference and then obtain the Controls[] property, which is an array of references to controls on the front panel.

3. Pass these control references into a For Loop and check whether the reference belongs to a control (*not* an indicator, because we do not want to disable indicators) and is not already disabled. If these conditions are met, then disable the control and build an array that is stored in a shift register. This array will store all of the controls that we have disabled.

Run your VI and press the `generate waveform` button. Watch the controls become grayed out while data is being generated and become re-enabled after the data generation is complete. Add more controls and indicators to the front panel and see that any control will be disabledyou don't have to write any additional code to handle the new controls!

Control Reference Types: The VI Server Class Hierarchy

When you configure a Property Node that is linked to a control, you might have noticed that there are horizontal dividers that group the various properties (the same is true of the grouping of the methods on Invoke Nodes). You will also notice that the top three groups of properties are common to all control types, but the groups below the first three depend on the specific type of control.

For example, Figure 15.58 shows the difference between the properties available for a generic control (left), a string control (middle), and a Boolean control (right). Note that all of these property lists have the top three groups of properties in common. However, the string control and Boolean control each have an additional group of properties that are specific to the string and the Boolean. This last group of properties is what makes them different from other types of controls.

Figure 15.58. Different control types add special properties to the "base" set of control properties common to all control types

[View full size image]

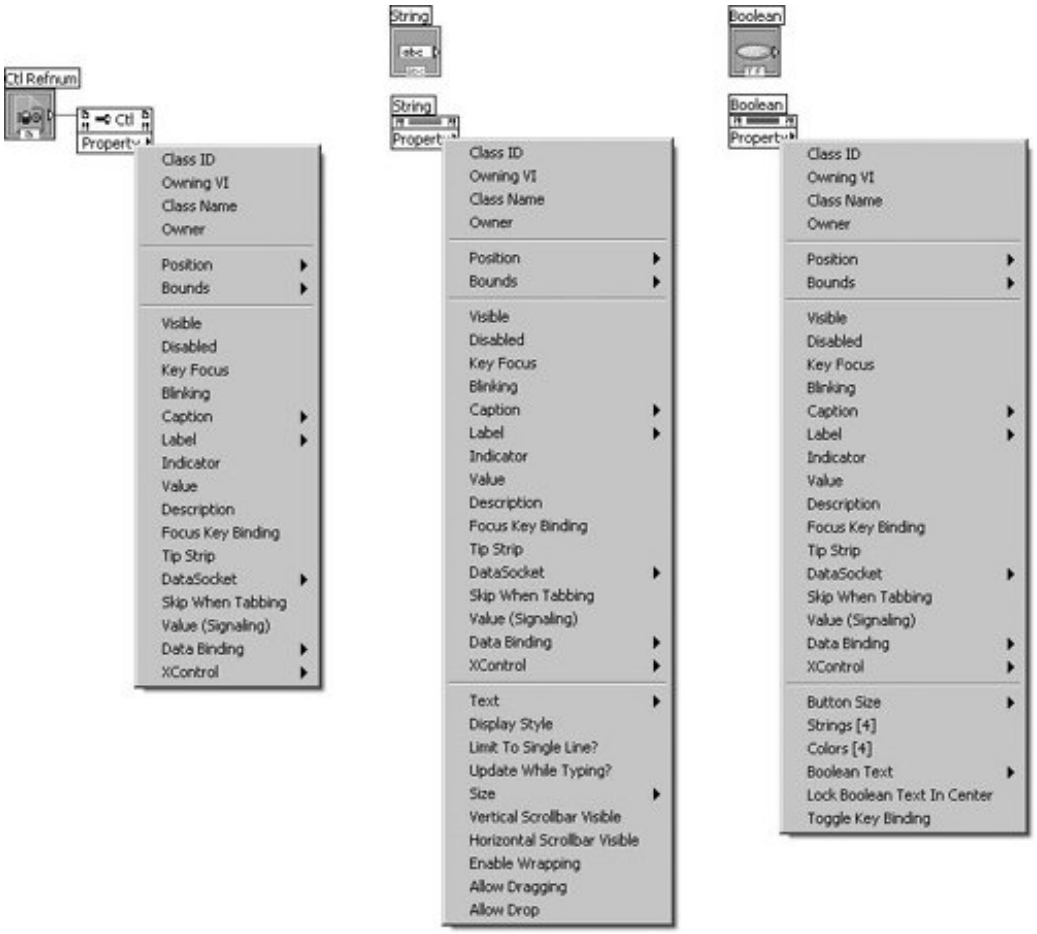


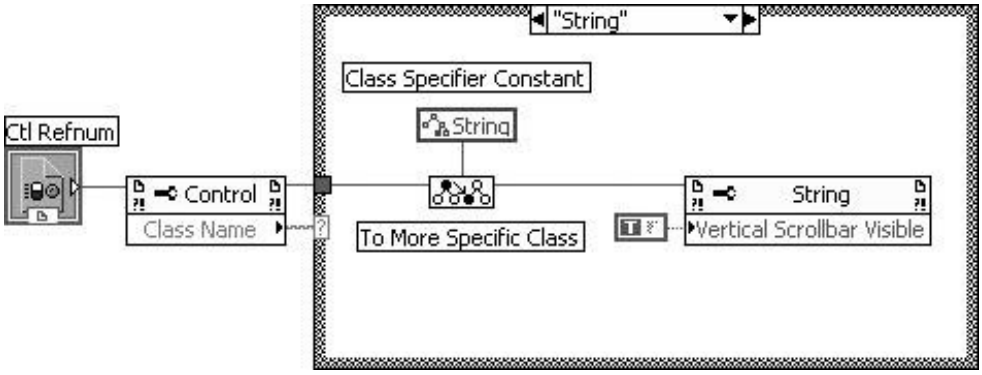
Figure 15.58 highlights another important concept. When you access the properties of a control using a generic control reference, you cannot access any of the properties specific to a certain type of control. This concept is similar in nature to the example we saw in the last section where the Value property read from a generic control is a variant (generic) data type. If we do not know what type of control we might be accessing the properties of, the list of properties available to us can contain only those properties that are common to all control types.

However, this does not mean that we cannot access those specific properties. We can first read the Class Name property to determine which specific type of control is contained in a generic control reference (as we read the Indicator property to select only controls in Figure 15.58). By reading the Class Name property to determine the specific type of control, we can then intelligently convert a generic control reference to the appropriate more specific type of control reference.

Figure 15.59 shows an example of this technique. Note that we first check to see if the control's Class Name property is "String." Then we use the To More Specific Class function to convert the generic control reference to a control reference of type String. Using the String control reference, we can access properties and methods that are specific to string controls, such as the Vertical Scrollbar Visible property.

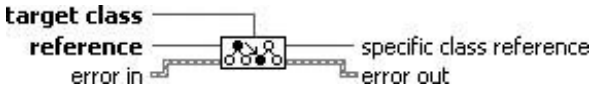
Figure 15.59. Intelligently converting a control reference type from

generically typed to more specifically typed, in order to access specific control properties



The To More Specific Class function (see Figure 15.60) is used to convert a control reference to a more specific type. If the conversion is invalid, this function will return an error (which we neglected in the diagram shown in Figure 15.59 for the sake of simplicity, but at our peril). For example, if you read the Class Name property and find that the control is a Boolean type, and then you try to use To More Specific Class to convert the control reference to a String type, an error will result because the operation cannot be performedyou cannot convert a Boolean control's reference into a String typed reference.

Figure 15.60. To More Specific Class



The To More Specific Class function needs to know which class to convert our reference into. To specify this, we wire a Class Specifier Constant (Figure 15.61) to the `target class` input of the To More Specific Class function. This will define the *type* of the `specific class reference` output. This concept is *very* similar to how both the Unflatten From String and Variant to Data functions each convert generically typed LabVIEW data into specifically typed LabVIEW data. (And, note that these two functions also have *type* inputs.)

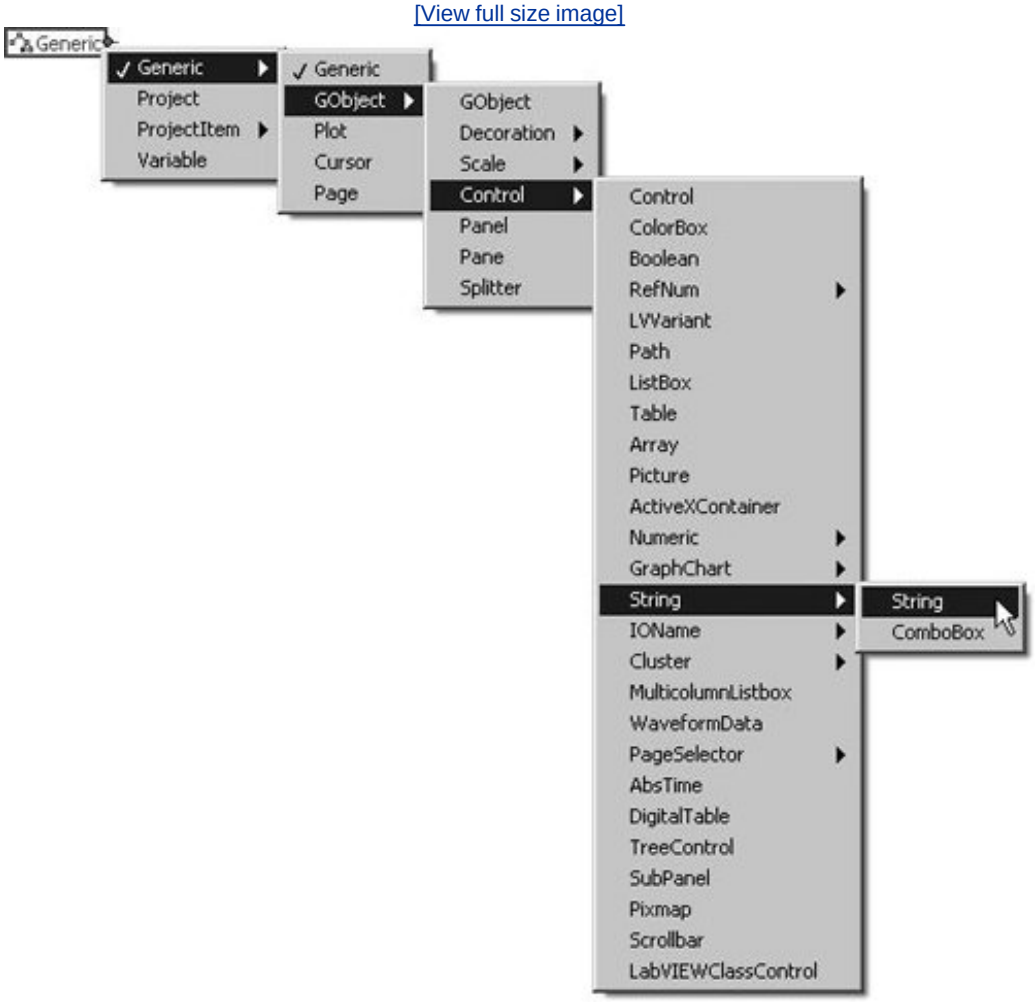
Figure 15.61. Class Specifier Constant

Mouse-click on the Class Specifier Constant using the Operating tool to select the class for the Class Specifier Constant. (You can also select a class from the Select VI Server Class pop-up

submenu.) As you can see in Figure 15.62, the String class can be found at the following submenu: Generic>>GObject>>Control>>String.

Figure 15.62. Setting the type of a Class Specifier Constant to a String control type

[View full size image]

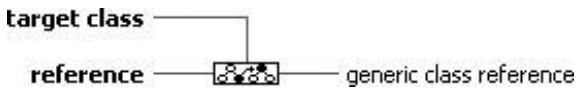


This menu structure of control types is organized in a *class inheritance hierarchy*. This is just a fancy way of saying that a String is a specific type of Control, which is a specific type of GObject, which is a specific type of Generic. As we drill down the menu, we are getting more specific, and as we move up the menu, we are getting more generic (less specific).

Conversion of control references to more generic types can be done using the To More Generic Class function. This works in exactly the same way as the To More Specific Class function with the exception that To More Generic Class does not have an error input or output (see Figure 15.63). Conversion to a more generic type *always* worksthere is no way to make a mistake. (Well, really all the mistakes can be caught while you're wiring, because the LabVIEW editor will enforce the rules.

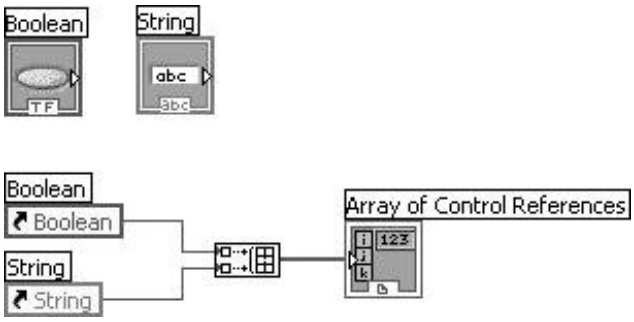
Try to convert a Boolean control reference into a Panel control reference and you'll get broken wires.) For example, the statements "a Boolean control is a control" and "a string control is a control" are always correct. Boolean and string controls, are by definition, controls. However, the statement "a control is a Boolean" makes sense if, and only if, the control actually is a Boolean control. There is a possibility that it is another type of control, so we must be careful when converting to more specific classes and always check the Class Name property before performing the conversion.

Figure 15.63. To More Generic Class



Conversion of specific control reference types to more generic control reference types does not need to be done explicitly using the To More Generic Class function. This conversion can also be done using the coercion feature of LabVIEW. For example, Figure 15.64 shows how we can wire two VI Server Reference nodes, which are linked to specific controls on the front panel, into a Build Array node. The wire types of the Boolean and string references are different but, since String and Boolean are each a different type of Control, the Build Array node coerces the reference types to the more generic Control type. Note the coercion dots on the Build Array function's input terminals. This is where the specific control references are being converted to generic control references. (The coercion is always to the most specific class that is common to all the elements.)

Figure 15.64. Building an array of different, specific control references results in an array of generic control references

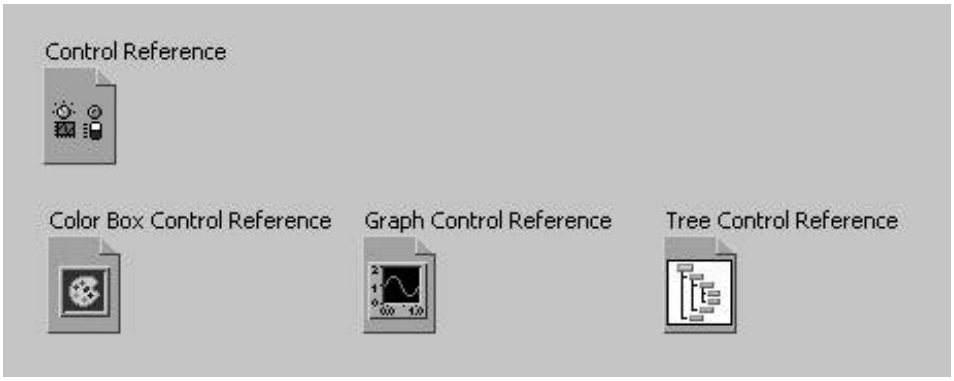


This is similar to how the Controls[] property of a VI's Panel outputs an array of generic

control references (which you saw earlier in the "Obtaining References to All Controls on a Front Panel" section of this chapter).

We can configure a Control Refnum (Modern>>Refnum palette) on the front panel to be specifically typed in the same way that we configured the type of the Class Specifier Constant. Pop up on a control reference control or indicator and select a class type from the Select VI Server Class submenu. When you change the class type, the control reference will change its appearance to reflect the control type, as shown in Figure 15.65.

Figure 15.65. Several Control Refnums of different types shown on a front panel



The Value Property (Is Evil): The Worst Kind of Global Variables

Yes, the value of a control is, technically speaking, a property of the controlbut you should try to avoid reading or writing to the Value property. The first reason is that it is much slower than passing data by wire to a control's terminal or local variable. But another, and possibly more important, reason to avoid using the Value property, is that it causes all of the benefits of data flow to be lost (and we chose LabVIEW for the benefits of data flow)! And, yes, we do remember that locals and globals also subvert data flow.

When we use control references as a way to get and set the value of controls in our application, we are simply using an insidious form of global variable (and the haphazard use of global variables is not recommended). But using control references in this way is far worse than using regular global variables because there is no way to know where the read or write operations are occurring in the codethey could be happening anywhere in your application. At least with global variables, we can find the *global references*the locations where the global variables are placed on the block diagram. (Well, strictly speaking, we could use the Find>>References pop up on the control and trace the wires to the offending Property Nodes, but that is no trivial task.)

Generally speaking, the standards for acceptable use of globals also apply to the use of the Value

property. An additional acceptable use of the Value property is in the initialization of control and indicator values at application startup (and resetting them at shutdown).

The Value (Signaling) Property: Generating Value Changed Events Programmatically

In addition to the Value property, all controls have a Value (Signaling) property. The two are identical with the exception that writing to the Value (Signaling) property also *generates a Value Change event* that can be captured by an Event Structure (which you learned about in Chapter 12), just as if the user had interactively changed the value of the control on the front panel.

It is recommended that you use the Value (Signaling) property only when you need to generate an event in response to a programmatic value change, just as if the user had changed the value on the front panel.

Activity 15-9: Building an Emergency Abort Utility

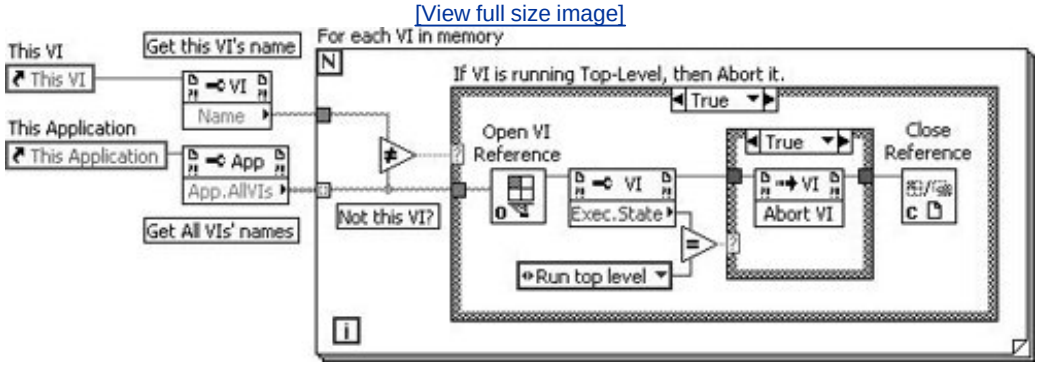
When debugging an application, sometimes you will get yourself into situations where you cannot access the Abort button to stop the application. In fact, you may have already found yourself in such situations.

You will now build Emergency Abort VIs, a utility that uses VI Server to abort all VIs in memory that are running top-level.

1. Open a new VI and build the block diagram shown in Figure 15.66.

Figure 15.66. Block diagram of the VI you will create during this activity

[View full size image]



2. Open the VI Properties dialog for this VI (from the File>>VI Properties menu item or the shortcut key <ctrl-I> [Windows], <command-I> [Mac OS X], or <meta-I> [Linux]), select the Execution category (drop-down list), and enable (check) the Run when opened checkbox.

3. Save this VI as `Emergency Abort VIs.vi` in your `MYWORK` directory.

4. Test your VI by first closing it, and then opening it again by double-clicking it from the desktop. LabVIEW automatically runs your VI, due to the Run when opened setting, and aborts all running VIs. Test Emergency Abort.vi again, but with other VIs running. Notice how the running VIs stop running.

When you get yourself into a jam and you cannot abort your application, often you will not be able to access the File>>Open menu option from any of your VIs. In this case, you will need to run Emergency Abort.vi by double-clicking it from File Explorer (Windows) or Finder (Mac OS X). On Linux, you will need to run the VI from the command line using the following commands:

```
cd "<folder containing emergency abort VI>"

/usr/local/lv80/labview -launch " Emergency Abort.vi "
```

Note that your LabVIEW install path might differ. Also note that the use of the launch option causes LabVIEW to check for an existing LabVIEW application instance already running on the same display. If an existing instance is running, the existing instance opens the VIs passed in the command line, and the new application instance quits silently. If an existing instance is not running, LabVIEW opens the VIs passed in the command line in the new instance.

Final Thoughts on VI Server

As we mentioned at the beginning of this section, don't worry too much if the VI Server seems perplexing or overwhelmingit's unlike anything you've encountered in LabVIEW before, and it breaks all the normal rules of how LabVIEW VIs interact with each other. Once you get comfortable with LabVIEW programming, you can always return to explore the numerous VI Server examples built into LabVIEW.

PREV NEXT

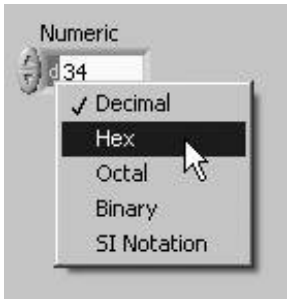
Radices and Units

A useful feature of LabVIEW numeric types is that you can manipulate them as more than just pure numbers. Normally, you represent numbers in decimal format. But for some applications, you may need to view numbers in a binary or hexadecimal representation. Similarly, you may want to treat certain numerical variables as having an associated unit (such as feet, calories, or degrees Fahrenheit)especially if you plan to do unit conversions. You can choose a different number base (called a radix) independently for any LabVIEW numeric control or indicator. That choice has no effect on the value of the underlying number; it only affects its visual display. You can also associate a unit with the control or indicator, and that unit will propagate into the actual number in the attached wire and be preserved, enforced, and appropriately modified by any math performed on the signal.

Radices

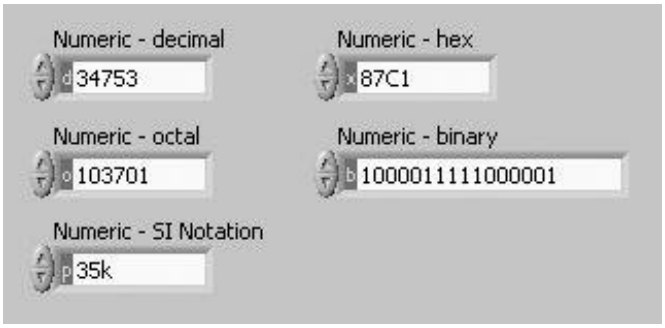
LabVIEW's numeric displays are very flexible. You can choose to display numbers in decimal, hexadecimal, octal, or binary format by selecting Visible Items>>Radix from the numeric's pop-up menu. Click with the Operating tool (do NOT pop up) on the tiny letter that appears, and you'll pull down a radix selection menu (see Figure 15.67).

Figure 15.67. Adjusting the radix of a numeric control



If your numeric is not an integer (DBL, for example), all radix options except for Decimal and SI Notation are disabled (see Figure 15.67).

Figure 15.68. Numeric controls having the same data, but configured with different radix displays



A user who is working with digital I/O, for example, might find it very useful to represent a numeric control in binary, because it would be easy to see the one-to-one correspondence of each digit to each digital line.

Units

Any floating-point numeric control can have physical units, such as meters or kilometers/second, associated with it. You can access the unit label by selecting Visible Items>>Unit Label from an object's pop-up menu.

Once the unit label is displayed, you can enter a unit using standard abbreviations such as `m` for meters, `ft` for feet, etc. (see Figure 15.69). If you enter an invalid unit, LabVIEW flags it with a `?` in the label. If you want to know which units LabVIEW recognizes, enter a simple unit such as `m`, and then pop up on the unit label and select Unit . . . . A dialog box will pop up, showing you information about LabVIEW's units (see Figure 15.70).

Figure 15.69. Selecting Build Unit String . . . from the pop-up menu of the units display launches the Build Unit String dialog

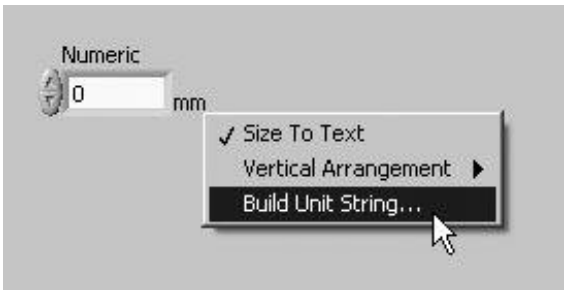
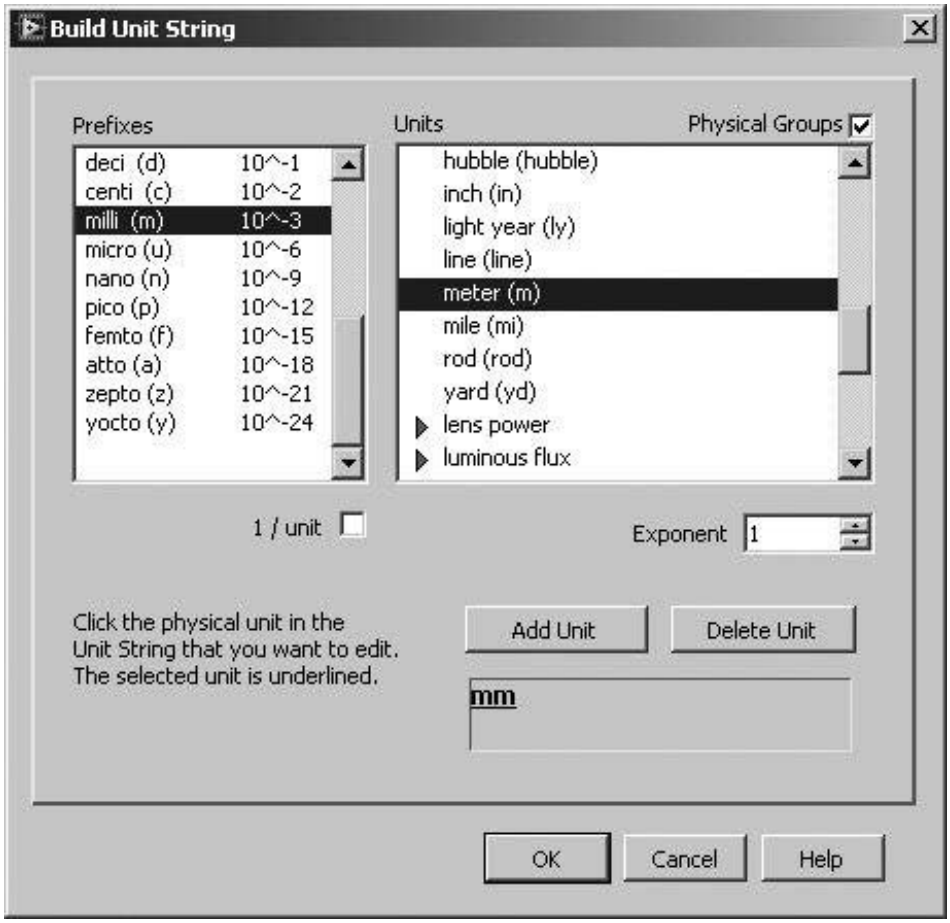


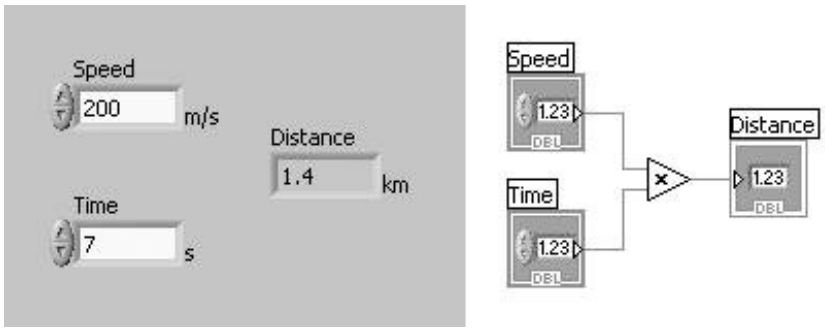
Figure 15.70. Build Unit String dialog



Any numeric value with an associated unit is restricted to a floating-point data type.

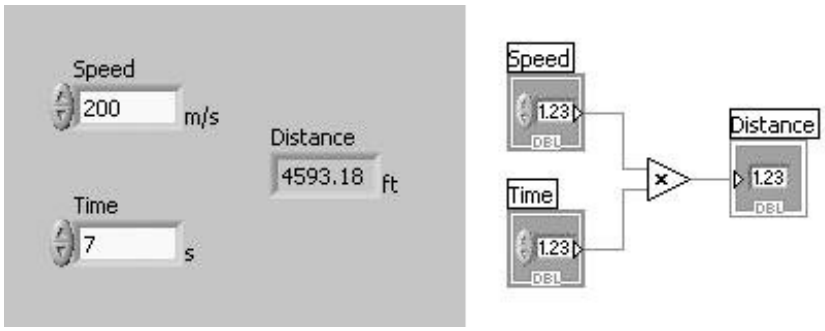
LabVIEW preserves the associated unit throughout any operations performed on the data. A wire connected to a source with a unit can only connect to a destination with a compatible unit, as shown in Figure 15.71.

Figure 15.71. Multiplying numbers that have units (output in km)



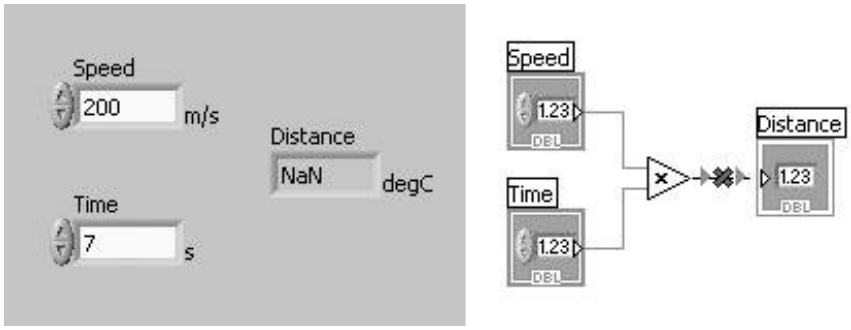
When we say "compatible," we mean that the destination unit has to make sense from a physical or mathematical point of view. The destination unit does not have to belong to the same classification system (e.g., SI versus English units), as shown in Figure 15.72. Thus, you can transparently convert compatible units from different systems in LabVIEW.

Figure 15.72. Multiplying numbers that have units (output in ft)



You cannot connect signals with incompatible units, as shown by the broken wire in the block diagram of Figure 15.73. If you try, the error window (discussed soon) will contain a block diagram error: `You have connected numeric data types that have incompatible units`.

Figure 15.73. Multiplying numbers that have units (output in degC does not work since it is not a unit of length)



Some functions are ambiguous with respect to units and cannot be used if a unit is assigned to the data you are trying to operate on. For example, if you are using distance units, the Add One function cannot tell whether to add one meter, one kilometer, or one foot.

Finally, you need to be wary of doing arithmetic that combines numbers with and without units. Well, not really wary, just aware. For example, you can multiply a unitless constant times a number with a unit, but you cannot subtract a number with a unit from a number without a unit. But you don't have to worry about thatLabVIEW enforces the rules with broken wires! Even better, it will prevent you from calculating force as mass times velocity because you forgot a factor deep in a twenty-step derivation.

LabVIEW allows you to "add" a unit to a unitless number using the Convert Unit function (from the Programming>>Numeric>>Conversion palette), shown in Figure 15.74.

Figure 15.74. Convert Unit: Converts a physical number (a number that has a unit) to a pure number (a number with no units) or a pure number to a physical number. To configure the units string, pop up on the Convert Unit function and select Build Unit String.

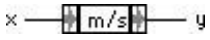
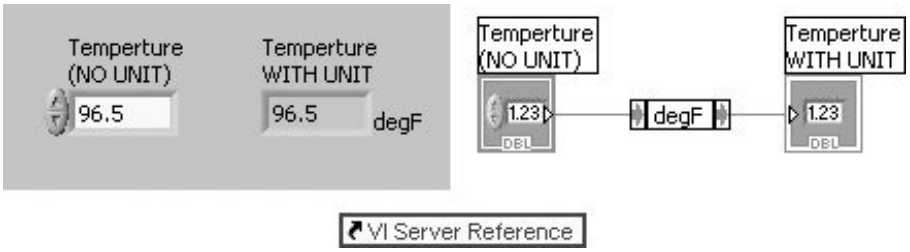


Figure 15.75 shows an example of how we can use the Convert Unit function to add units to a number.

Figure 15.75. Converting a pure number (no units) to a physical number (with units) using the Convert Unit function



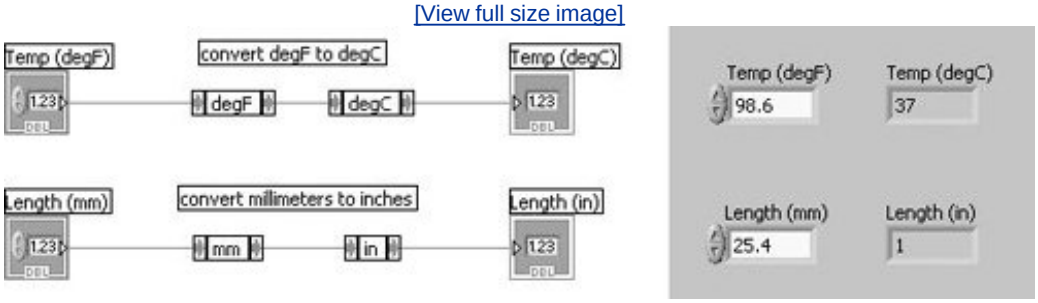
To specify the unit, just type the unit inside the middle box of the terminal or, as with numeric types, pop up on the terminal and select Build Unit String . . . for a list of valid units. The Convert Unit function also allows you to "strip" units from a number (convert a number with units to a pure number).

Simple Unit Conversion: A Great Trick to Know

One very powerful trick that every LabVIEW developer should know is that you can quickly convert the units of a pure number (one without any units) by cascading two Convert Unit functions. Figure 15.76 shows two examples of this.

Figure 15.76. Using two Convert Unit functions is an easy way to convert the units of pure numbers.

[View full size image]



Configure the upstream Convert Unit function with the incoming pure number's units and the downstream function with the outgoing pure number's units. It's as easy as that!

Automatically Creating a SubVI from a Section of the Block Diagram

One key to writing your program right the first time with the least amount of bugs is *modularity*. The modules in LabVIEW are, of course, subVIs. When you're writing a large application, it's essential to break up the tasks and assign them to subVIs. If you didn't plan ahead (which is by far the best approach) or if the scope of the project increases unexpectedly (never happens), LabVIEW provides a nice recovery mechanism. The Edit>>Create SubVI menu option makes converting sections of your block diagram into subVIs much simpler. If you're working on some code and decide you should make part of it a subVI, no problem! Select the section of a VI's block diagram, as shown in Figure 15.77, and choose Edit>>Create SubVI. LabVIEW converts your selection into a subVI and automatically creates controls and indicators for the new subVI. The subVI replaces the selected portion of the block diagram in your existing VI, and LabVIEW automatically wires the subVI to the existing wires, as shown in Figure 15.78. You'll probably want to add a few features and move a few terminals around to match the wiring elsewhere in your project and, of course, there are terminal descriptions to add, but LabVIEW has given you a good start. This feature is also an excellent option to use, for example, if you need to repeat part of your block diagram in another VI. If two places need the same code, shouldn't it be in a subVI?

Figure 15.77. Selecting a block of code, which you wish to convert into a subVI

[View full size image]

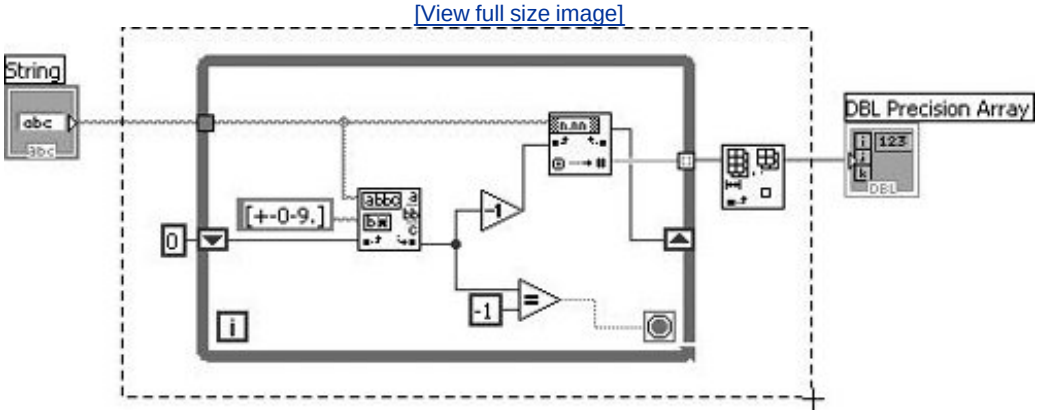
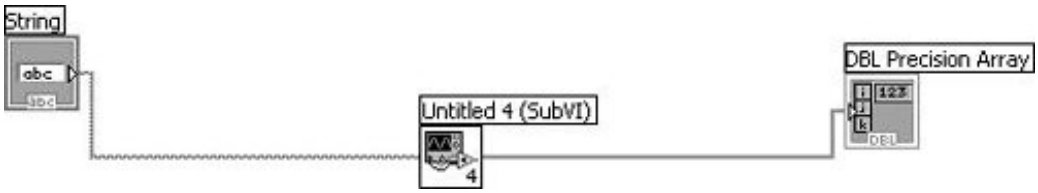


Figure 15.78. The subVI that replaced your selected block of code after selecting Edit>>Create SubVI from the menu

[View full size image]



As with all menu options, you can create a custom shortcut key for the Create SubVI command in the Menu Shortcuts section of the Tools>>Options . . . dialog, as we discussed earlier.

Despite how easy it is to use, let's look at some rules for using the Create SubVI option.

Because converting some block diagrams into subVIs would change the behavior of your program, there are some instances where you cannot use this feature. These potential problems are listed next.

- You cannot convert selections that would create a subVI with more than 28 inputs and outputs, because that is the maximum number of inputs and outputs on a connector pane. We'd feel sorry for the user who would have to wire even 15 inputs! In a case like this, select a smaller section or group data into arrays and clusters before selecting a region of the block diagram to convert.
- You cannot convert selections in which items inside and outside a structure are selected, yet the structure itself has not been selected.
- Linked Property nodes (those that are implicitly linked to controls on the front panel and do not have a reference input or output) that are contained within the selection of code, will be replaced by an unlinked Property Node. To maintain the connection between the control and the Property node, a VI Server Reference to the control will appear on the block diagram and the reference will be passed into the new subVI (and subsequently passed to the Property

node) via a control reference data type. Figures 15.79 and 15.80 show the before and after state of this operation.

Figure 15.79. Before: select a linked Property Node

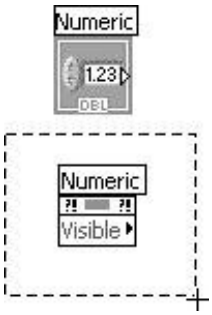
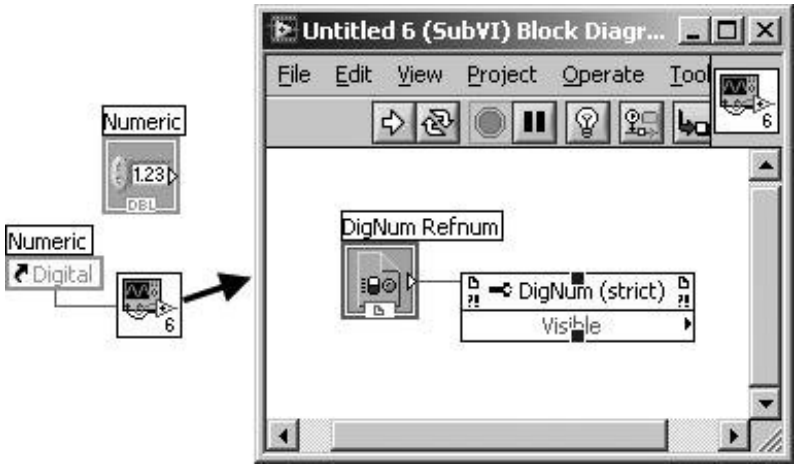


Figure 15.80. After: selected property converted into subVI



- When your selection includes local variables but does not include the corresponding control, one local variable for the control remains in the calling VI and passes the data to or from the subVI. Within the subVI, the first local variable for a control becomes a read or write for the control and subsequent local variables refer to the control in the subVI. Yikes! What does that mean? Bottom line: Be very careful when you create a subVI from the selection if it includes locals.
- When your selection includes local variables or front panel terminals inside a loop, the value that they are measuring may be changed elsewhere on the block diagram as the loop runs. Thus, when you convert the loop into a subVI, there is a possibility that the functionality of the selected code changes. If you have selected some but not all of the local variables or front panel terminals, LabVIEW displays a warning that allows you to choose between continuing and canceling the operation.

The Create SubVI option can be a really handy tool in cases where you start with what you thought was a small project (hah!), and you soon realize your program is going to be more complex than you originally thought. However, don't just squeeze a bunch of diagram mess into a subVI to increase

your workspacesubVIs should always have a clear, well-defined task. When considering whether to make a part of your diagram a subVI, you might ask yourself, "Could I ever use this piece of code or function somewhere else?" We'll come back to subVIs and modular programming again in Chapter 17, where we look at good programming techniques.

PREV NEXT

A Few More Utilities in LabVIEW

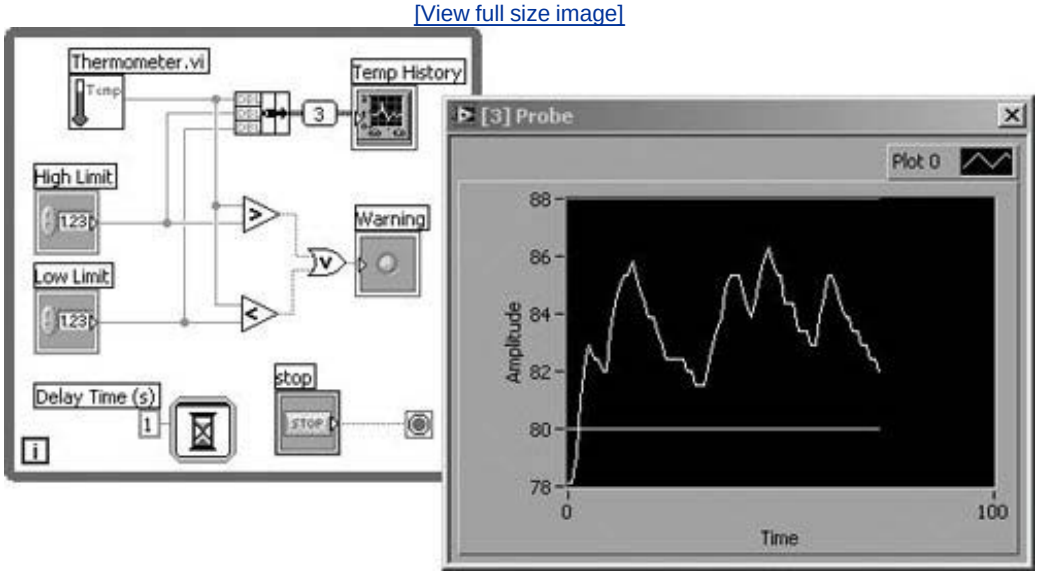
Custom Probes

In Chapter 5, "Yet More Foundations," you learned about probes, which allow you to peek inside the value of a wire while a VI is executing. If you remember, you can right-click on a wire and select Probe to get a floating window that displays this value.

LabVIEW allows you to take this debugging tool one step further with custom probes. You can probe using any conventional indicator by selecting Custom Probe> from the wire's pop-up menu, and then choosing the desired indicator with which to probe. For example, you could use a chart to show the progress of a variable in a loop, because it displays past values as well as current ones (see Figure 15.81). LabVIEW won't let you select an indicator of a different data type than the wire.

Figure 15.81. Custom probe with a chart

[View full size image]



Remembering Probe Values

Retain Wire Values Button

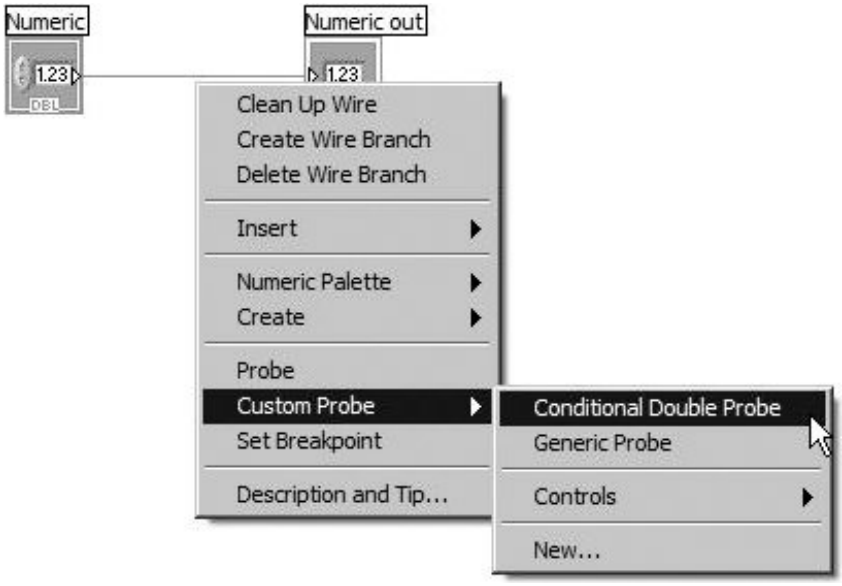
Normally when you first use a probe, it will not display any data until you run the VI *and* data flows through the wire you are probing. However, sometimes you may want to immediately obtain the most recent value of the data that passed through the wire without having to run the VI.

This is called "retaining wire values" and you can enable this easily by pressing the Retain Wire Values button on the VI's toolbar. You'll need to have this option turned on before you run the VI for the probe to be able to display data from the last time the VI ran. But be aware that this option will cause your VI to require more memory in order to hold data for all of the wires.

Using and Creating Custom Probes

Custom probes are a powerful debugging toolthey act as both probes and conditional breakpoints (allowing you to choose the condition that suspends VI execution). You can use custom probes by right-clicking on a wire and selecting Custom Probe> from the shortcut menu, as shown in Figure 15.82. The list of items at the top of the Custom Probe> submenu (above the first divider) are all custom probes. LabVIEW only displays the custom probes that will work with the data type of the wire that you right-clicked on. LabVIEW ships with some useful custom probes for most LabVIEW data types. As you can see in Figure 15.82, the Conditional Double Probe can be placed on a wire that is a DBL type.

Figure 15.82. Inserting a Conditional Double Probe onto a DBL numeric wire



After we select Conditional Double Probe, a probe instance will be created in a new floating window, as shown in Figures 15.83 and 15.84. The Conditional Double Probe has a tab control with a

Data tab and a Condition tab. The Data tab displays data as it passes through the wire, just like a regular probe. The Condition tab allows you to specify a condition that will cause the VI to be suspendedthus becoming a conditional breakpoint. This is a very powerful feature.

Figure 15.83. Data tab of the Conditional Double Probe

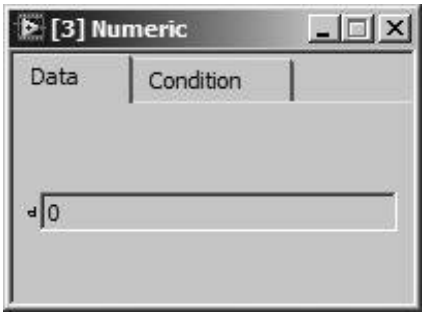
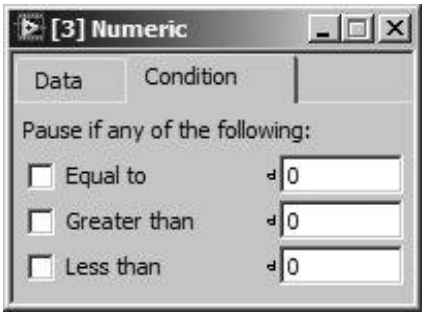


Figure 15.84. Condition tab of the Conditional Double Probe



Other data types have equally useful custom probes. For example, the Conditional Error Probe (shown in Figure 15.85) allows you to probe error clusters and suspend execution when certain errors occur. The Conditional String Probe (shown in Figure 15.86) allows you to probe strings and suspend execution under a variety of possible conditions.

Figure 15.85. Conditional Error Probe

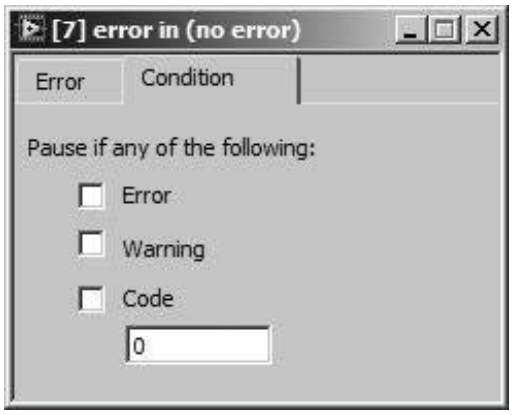
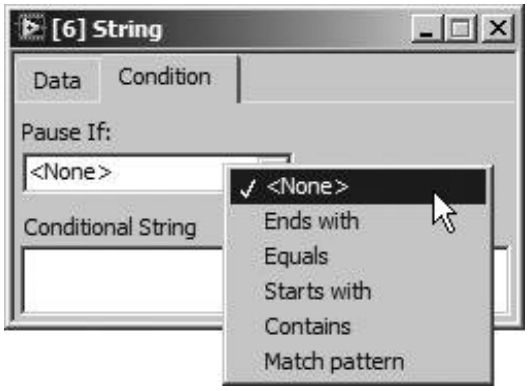


Figure 15.86. Conditional String Probe



While extremely powerful, custom probes are nothing fancy. They are regular LabVIEW VIs having one input for the probed wire's data, one Boolean output for the conditional breakpoint, and a connector pane with only two terminals, as shown in Figure 15.87. For LabVIEW to find your custom probes and make them available from the Custom Probe> submenu, you must place them inside the `Probes` directory beneath the Default Data Directory. (The Default Data Directory may be set in the Paths section of the LabVIEW options dialog.)

Figure 15.87. Connector pane of a custom probe VI

When placed on a wire, LabVIEW creates a separate instance of the custom probe VI. Each time data flows through the wire, the custom probe VI instance is run. If the custom probe VI instance returns a Boolean value of TRUE, then execution will be suspended, just as if there were a breakpoint on the wire.

A custom probe should (in nearly all circumstances) execute and then return immediately. It should not have a while loop, or other code that could take a long time to execute. Rememberyour main code is not running while the custom probe is running.

When LabVIEW populates the Custom Probe> submenu with the list of custom probes inside the Probes folder, it only shows the custom probes with an input data type that exactly matches the data type of the wire.

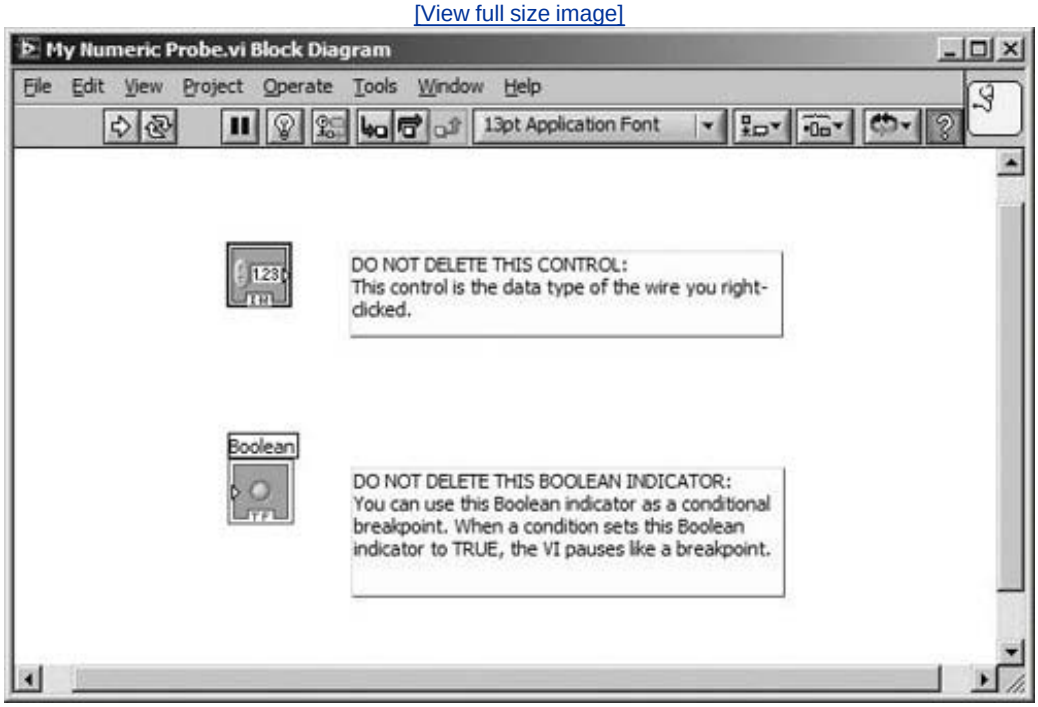
To create a new custom probe, right-click on a wire and select Custom Probe>>New from the shortcut menu. LabVIEW will present you with the Create New Probe dialog. From here, you can choose to Create a probe from an existing probe or Create a new probe. Make sure that you save the new custom probe VI in the suggested location, so that LabVIEW can find it later, when you right-click on wires.

When you open the new custom probe VI, you will see a block diagram that looks similar to Figure 15.88. Do not delete these controls. These are necessary for the proper operation of your custom probe. In the block diagram, you can perform any operations that you wish. You can customize the front panel to display the data, and provide the user with choices about how to control the conditional

breakpoint.

Figure 15.88. Block diagram of a newly created custom probe

[View full size image]



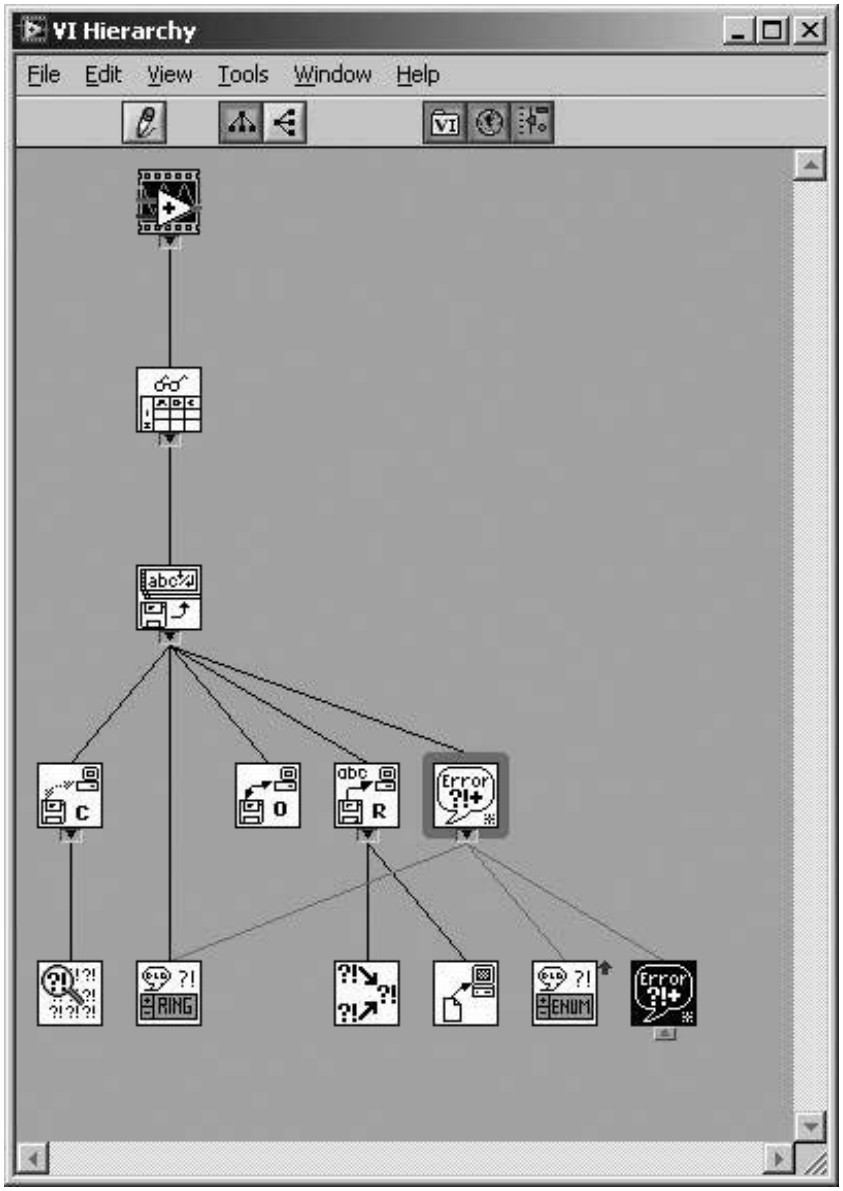
To get a good understanding of what is possible with custom probes, take a look at the existing custom probes that ship with LabVIEW by choosing Create a probe from an existing probe from the Create New Probe dialog.

The VI Hierarchy Window

Generally, you will find the VI Hierarchy window useful when you are working on a VI of fair to high complexitysuch as one that contains 10 or more subVIs. The VI Hierarchy window can help keep track of where subVIs are called and who calls them.

With a VI's front panel open, select View>>VI Hierarchy to bring up the VI Hierarchy window for that VI. The VI Hierarchy window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and globals (see Figure 15.89).

Figure 15.89. VI Hierarchy window



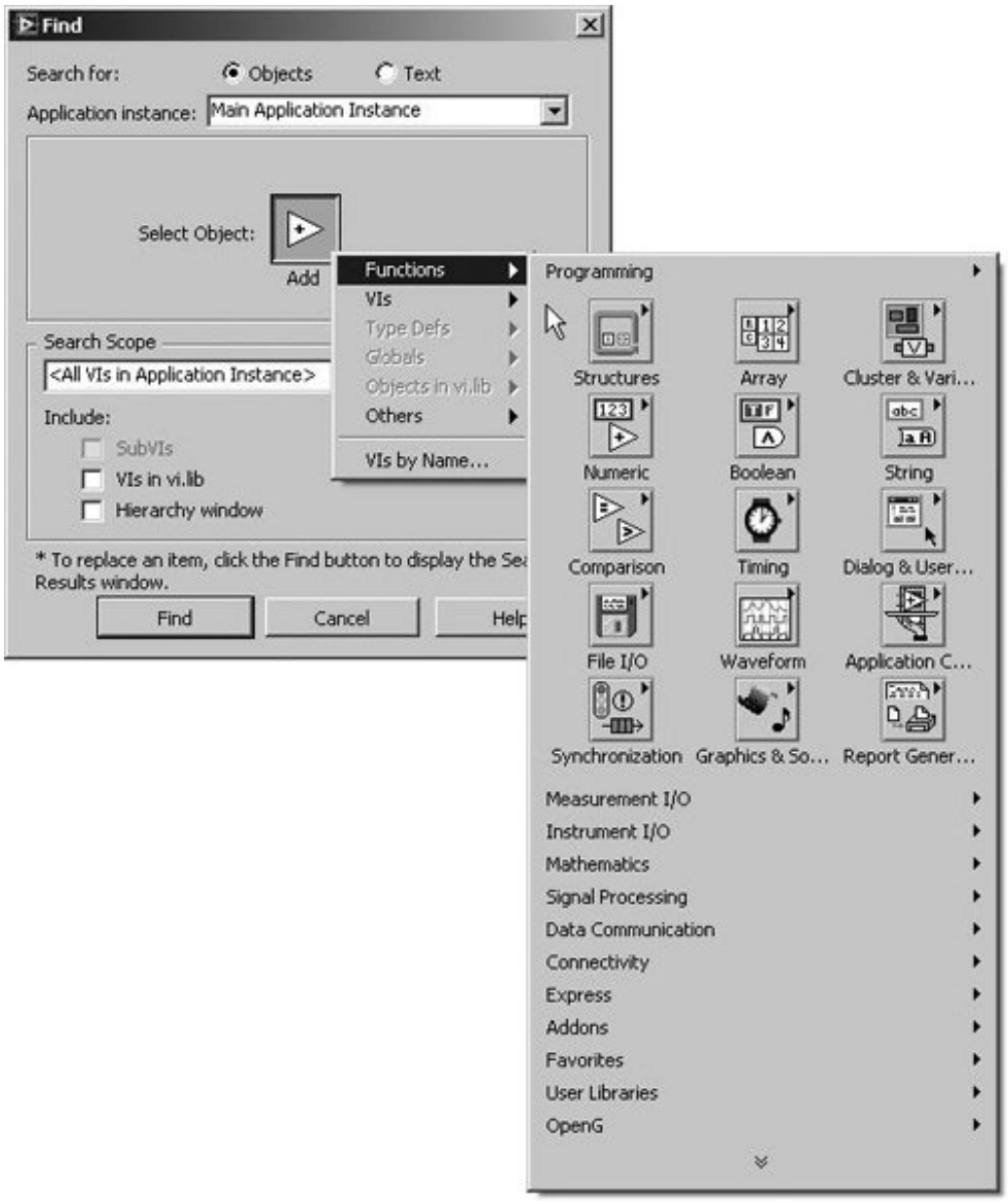
The buttons in the VI Hierarchy window's toolbar let you configure a number of aspects of the display. For example, the layout can be displayed horizontally or vertically; you can hide or show as many levels of hierarchy as you wish; and you can include or exclude globals or type definitions. A nice feature in this window is the one that allows you to double-click on a subVI icon to open its front panel.

Searching for Objects in the Virtual Haystack

LabVIEW has a good search engine that you can use to quickly find and replace text, objects, or VIs in your project. LabVIEW's Find dialog, available by selecting Edit>>Find . . . from the menu, can quickly locate any LabVIEW function, subVI, global variable, attribute node, or text in your VIs. You can limit your search scope to a single VI, a VI and its subVIs, or a set of VIs, or include all VIs in memory (see Figure 15.90).

Figure 15.90. The Select Object button on the Find dialog allows you to search for any LabVIEW function or VI.

[View full size image]

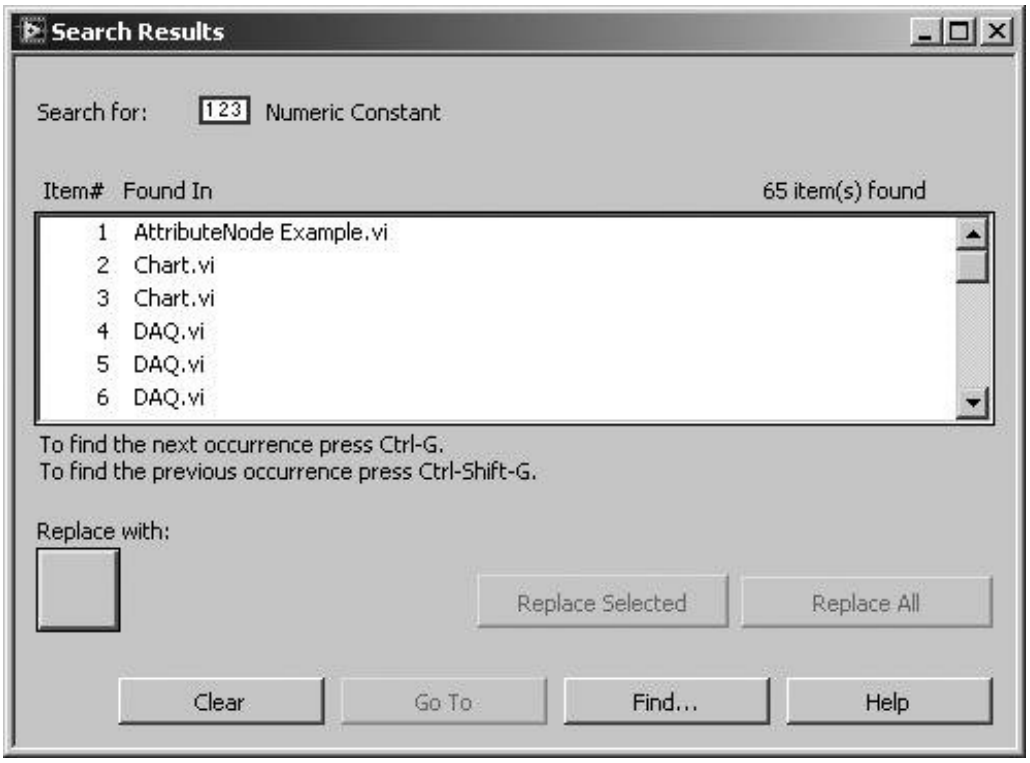


You can choose to search for objects or text by clicking in one of the Search for checkboxes. If you select Objects, click on the Select Object button to access the Select Object pop-up menu that allows you to choose the type of object you want to search for. If you select Text, you can type in the text, and click on the More Options . . . button to further limit the scope of the search to sections of VIs and object labels.

After you press the Find button, LabVIEW will display the Search Results dialog box, as shown in Figure 15.91.

Figure 15.91. The Search Results dialog shows a list of items that were

found during the search.



You can double-click on any item displayed in the Search Results window to have LabVIEW show you where that item resides or use <ctrl-G> (Windows), <command-G> (Mac OS X), or <meta-G> (Linux) to view each one in order (<ctrl-shift-G> (Windows), <command-shift-G> (Mac OS X), or <meta-shift-G> (Linux) to view in reverse order).

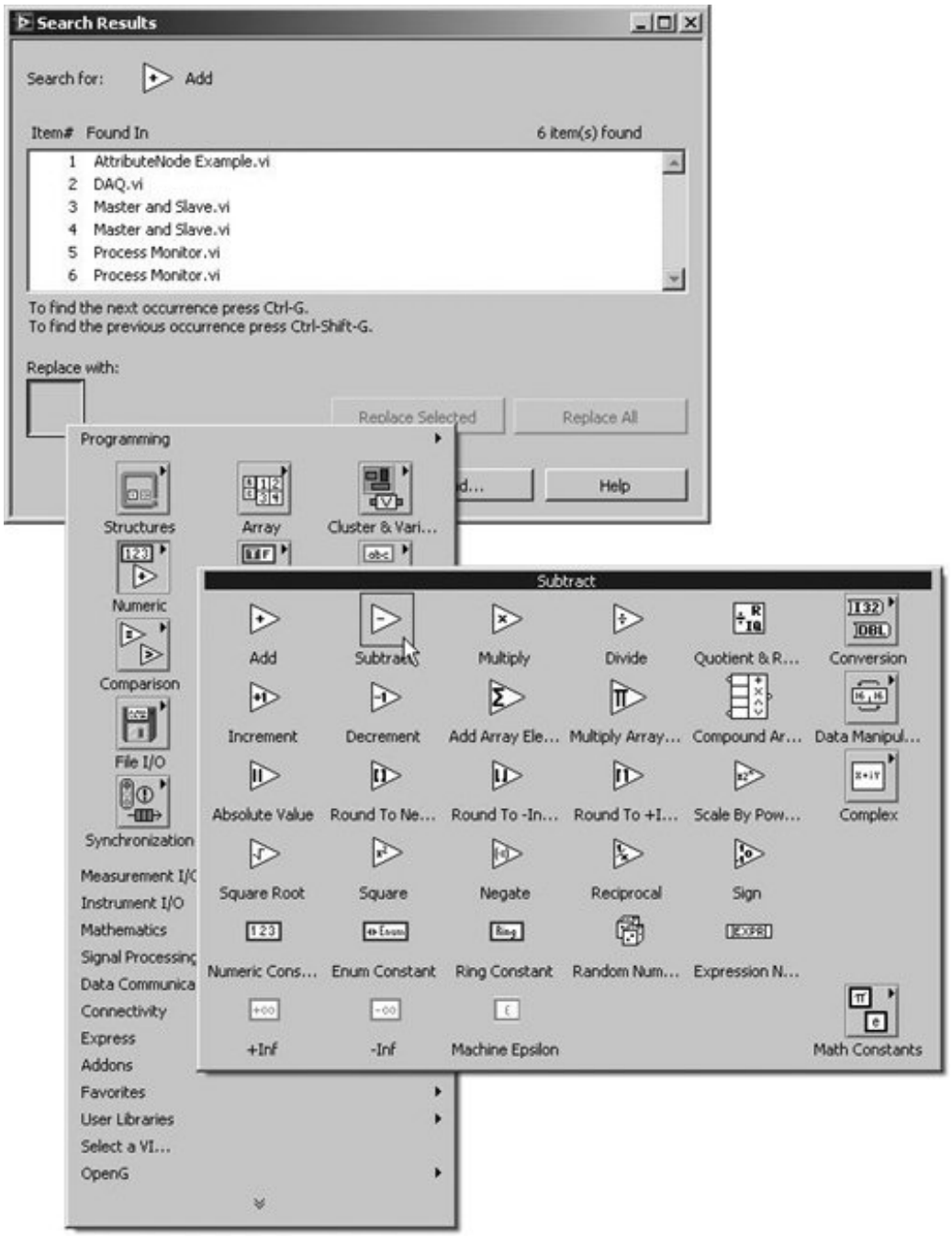
A very fast way to use the search tool is to pop up on a VI's icon and select Find All Instances. This will initiate a search and the Search Results window (shown in Figure 15.91) will appear with the list of found items. Note that you can access the Find All Instances option from the pop-up menu of subVIs, VIs shown in the VI Hierarchy window, and the VI icons located in the upper-right corner of VI front panels and block diagrams.

Replacing Search Result Items

You can choose to replace individual or all search result items. For object searches, you can replace result items with functions or VIs that you can select from the Replace With: pop-up menu (see Figure 15.92). For text searches, Replace with: is a text box, where you can specify the text with which to replace the search result items. Pressing the Replace Selected button will replace the selected item in the results list, and pressing the Replace All button will replace all items in the results list.

Figure 15.92. The Replace with button on the Search Results dialog allows you to specify a function or VI that you wish to replace for the items found in the search.

[View full size image]

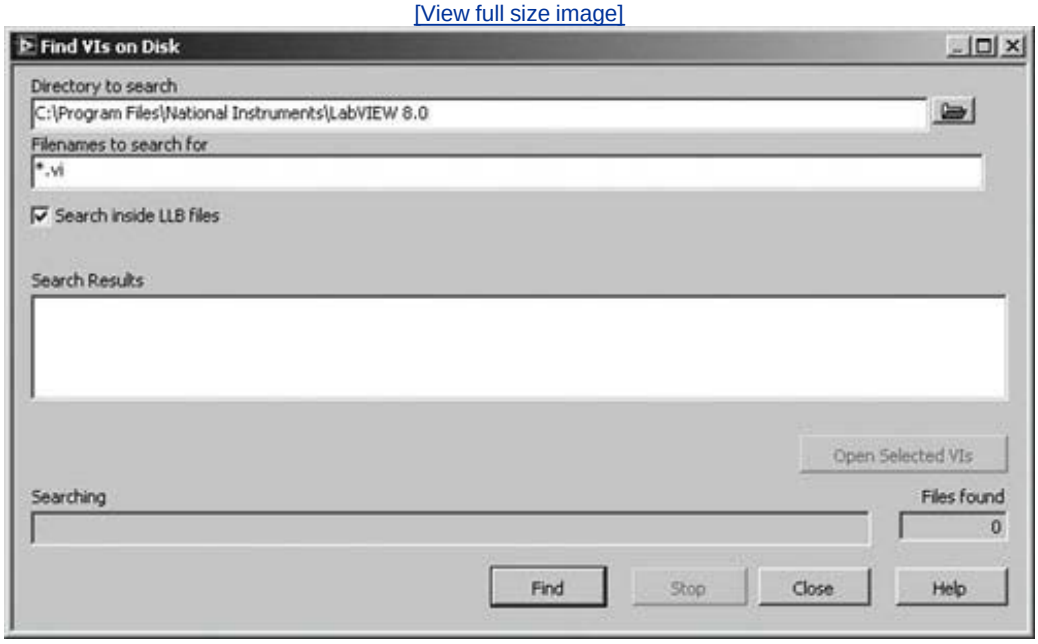


Find VIs on Disk

The Find VIs on Disk dialog is available by selecting Tools>>Find VIs on Disk . . . from the menu (see Figure 15.93). This provides a very nice way to search for VIs beneath a directory on disk. The main advantage of this tool over the file search tools built into your operating system is that it has the ability to search for VI *inside* LabVIEW LLB filesthe operating system cannot see inside LLB files.

Figure 15.93. Find VIs on Disk dialog

[View full size image]



More Tools

Under the Tools menu, you will find an assortment of other LabVIEW utilities, such as the Web Publishing Tool (which creates static or interactive web pages from your VI), a VI Profiler (which measures the time and memory a VI consumes), and more. We'll discuss some of these tools in the upcoming chapters.

Depending on the version of LabVIEW you have installed (evaluation, basic, professional, full development) and any add-on toolkits you may have (Internet Toolkit, Database Connectivity Toolkit, etc.), you will see the corresponding utilities in this menu.

PREV NEXT

Wrap It Up!

This chapter covered a potpourri of interesting and useful LabVIEW features. We covered aspects such as the Options, VI Properties, and SubVI Node Setup options, the VI Server, key focus, radices, units, creating subVIs from selections, the VI Hierarchy, and the Find function.

The Options . . . command allows you to control a myriad of options under several categories. These options give you control over the LabVIEW environment and let you personalize aspects such as fonts, colors, printing options, paths, and so on.

VI Properties and SubVI Node Setup contain useful selections to customize the aspects of a particular VI at the top-level or as a subVI. With VI Properties, you can hide or show the Toolbar buttons, center the window, control execution options, and so on. The SubVI Node Setup allows you to "pop up" the window of a subVI as it's called and close it afterwards.

The VI Server was introduced as a powerful object-oriented approach to controlling VIs and LabVIEW. You saw how you can use the Property Node and Invoke Node to work with methods and properties of the LabVIEW Application class, VI class, and Control class.

Key focus is a Boolean attribute of a control that decides whether the control is "selected" and ready for input from the keyboard. You can set the key focus programmatically, or simply tab through all the controls. You can also assign special keys, such as function keys to toggle a particular Boolean control.

Radices and units are special built-in features of LabVIEW numeric displays. A numeric control, indicator, or constant whose radix is shown can be changed to a decimal, binary, octal, or hexadecimal representation. The optional built-in units on numeric types allow you to perform mathematical operations on numbers with units and provide automatic conversion for different unit systems.

The modularity of LabVIEW is highlighted by the ability to select any part of the block diagram and, under certain conditions, turn it into a subVI with the appropriate inputs and outputs.

You also learned how to create custom probes to help you debug your data.

Finally LabVIEW provides some advanced programming tools, such as a VI Hierarchy window, a Find utility that lets you search (and replace) functions and subVIs, and more.

16. Connectivity in LabVIEW

Overview

In this chapter, you'll learn about the different ways you can connect LabVIEW VIs to the Internet, your network, and other programs. We'll see how you can easily use LabVIEW's built-in web server to publish front panels to the Web. You'll become familiar with network variables and how they can be used to share data between LabVIEW VIs across the network. XML, ActiveX, .NET connectivity (Windows), AppleEvents, and pipes provide you with a way to have LabVIEW interact with external programs. We'll see where LabVIEW fits in the big picture of the "enterprise" data management and databases. And, finally, we will take a quick look at the report generation VIs.

Goals

- Know how to use the LabVIEW web server to publish VIs to the Web
- Learn how to send emails from LabVIEW
- Use shared variables to share data
- Find out how LabVIEW works with XML schema and XML data
- Discover the .NET, ActiveX, AppleEvents, and pipes for communicating with other applications
- Become familiar with LabVIEW's lower-level networking capabilities: TCP and UDP VIs
- Understand the role of databases
- Examine LabVIEW's capabilities to generate reports

Key Terms

- Internet
- Client-server
- URL
- HTTP
- Email
- Web server
- Shared variable
- DataSocket
- ActiveX
- .NET
- AppleEvents
- Pipes
- TCP/IP
- UDP
- Enterprise
- XML
- Databases
- Report generation VIs

Your VIs on the Web: The LabVIEW Web Server

Often you'll want other people to be able to access your data or your VIs through a web browser. When you allow others to access your VIs over the Web, you should determine in advance if you want people to just monitor the VI (read-only) or control the VI (send data and events back) from the web browser.

LabVIEW makes publishing your VIs to the Web easy with the built-in LabVIEW web server. With the LabVIEW web server, you can *dynamically* create web pages with images of a VI's front panel on the fly, without the need for any special coding in your block diagram. In this next section, you'll see how simple it is to set up.

Configuring LabVIEW's Built-in Web Server

Don't confuse the VI Server with the web server in LabVIEW. They're unrelated and independent of each other. The web server is for allowing users to view and control a VI remotely (as well as view/download HTML and other files) using a web browser. The VI Server is for letting local and remote VIs call methods and properties of your VIs.

To enable the LabVIEW web server, do the following:

1. To avoid system conflicts, make sure no other web server (e.g., Apache or IIS) is running on the same port before you enable the LabVIEW web server. If you don't know, most likely you do not have another web server running.

2. If you have a firewall installed, or are using Windows' built-in firewall, you may need to ensure port 80 is not blocked. (Port 80 is the default port the web server runs on.)

3. In LabVIEW, go to Tools>>Options>>Web Server:Configuration (as shown in Figure 16.1).

Figure 16.1. Web Server: Configuration options

[View full size image]

4. Check the box "Enable Web Server."

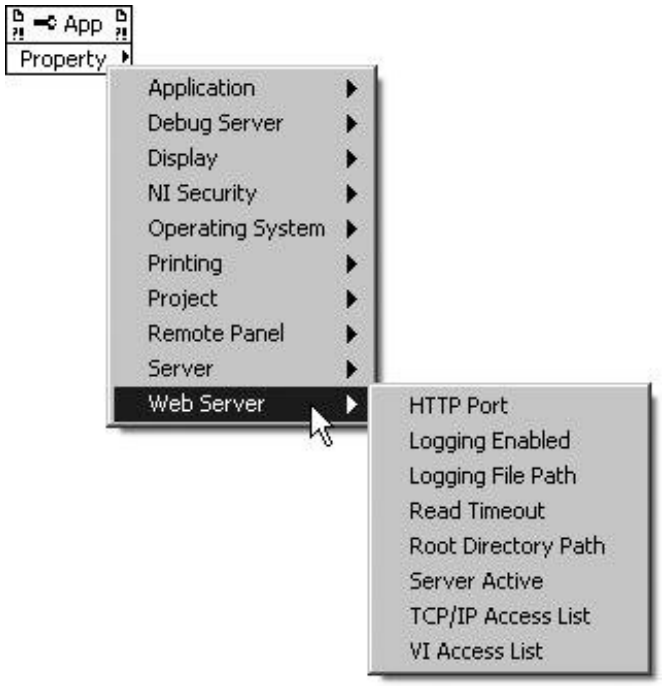
5. If you desire, change the root directory. This is the directory on your hard drive where your web documents will reside.

6. Click OK.

You can also control LabVIEW web server properties through the VI Server interface, using the property node for the Application class. This means you can programmatically turn

the server on or off, set permissions for the access list, and so on (see Figure 16.2).

Figure 16.2. Web Server properties accessible via an Application Property Node



Publishing to HTML with LabVIEW's Web Server

With the LabVIEW web server, there is nothing to program on the block diagramall the details are handled by LabVIEW. The VI that is to be displayed in a web browser must already be loaded into memory (in LabVIEW) to be served by the LabVIEW web server. The LabVIEW web server can publish your VI to the Web in one of three ways:

- "Snapshot" A static ("snapshot") image of a VI's front panel.
- "Monitor" An image of a VI's front panel that you can configure to auto-refresh every N seconds.
- "Embedded" A controllable version of the VI in the front panel. This option uses a browser plug-in to display the VI in real time and allows users to control the VI.

The following activity shows how easy it is to view the LabVIEW Example VI, Temperature System Demo.vi, through a web browser.

Activity 16-1: Using LabVIEW's Built-in Web Server to Publish Images and Animations

In this activity you will use LabVIEW's built-in web server to publish images and animations of a VI's front panel and view them in a web browser.

1. Make sure you've turned on the built-in web server, as described earlier.

2. Open the VI Temperature System Demo.vi (from the LabVIEW examples, in `/examples/apps/tempsys.llb`).

3. Open your web browser.

4. To see a static image of the VI, type in your web browser (on the same machine):

   http://127.0.0.1/.snap?Temperature+System+Demo.vi

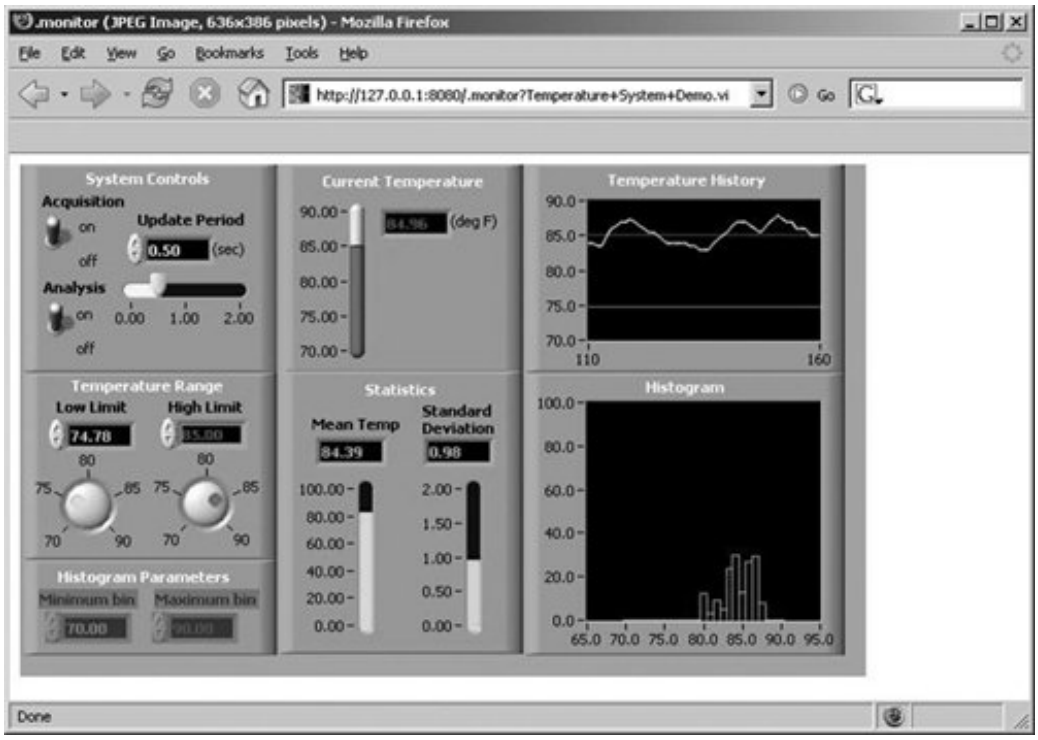
5. To see an auto-refreshing image, you will need to use a browser that supports "push" features on images. With Netscape's browser, type this URL:

   http://127.0.0.1/.monitor?Temperature+System+Demo.vi

You will see the image animate features like the graph. The push feature only works well on Mozilla Firefox, and not on Internet Explorer. On Internet Explorer, the whole page will auto-refresh continuously, instead of displaying pixel animation (see Figure 16.3).

Figure 16.3. An animated image of Temperature System Demo.vi viewed in a web browser

[View full size image]



Let's examine the URL from the previous activity a little more closely:

`127.0.0.1`

This is just the special IP address for the local machine (if you were viewing the VI over the web from a remote host, you would have to type in the real IP address, of course).

`.snap?`

The `.snap` part of the URL is a special command that tells the LabVIEW web server to take a snapshot of the VI that will follow the `?` character.

`.monitor?`

The other option, `.monitor`, is a special command that tells the LabVIEW web server to monitor and provide an animated image of the VI that will follow the `?` character.

`Temperature+System+Demo.vi`

This is just the URL encoding for "Temperature System Demo.vi." In any URL, spaces must be replaced with the "+" sign or "%20" and any special characters (e.g., "?", "/") must use the appropriate URL encoding.

Note that these dynamic documents returned by the LabVIEW web server are simply front panel images; they are not HTML documents. However, it is very straightforward to create your own HTML documents that embed dynamic images of a VI's front panel; you do this by simply specifying the URL as the image source in an <IMG> tag.

Activity 16-2: Using LabVIEW's Built-in Web Server to Publish Interactive VIs

In conjunction with the LabVIEW web server, you can use the simple Web Publishing Tool (from the Tools menu). This little utility builds a simple HTML file for you where you can add text around your VI image. In addition to creating HTML files, it exposes the functionality of how to embed interactive VI panels into your HTML document.

1. Make sure you've turned on the built-in web server, as described earlier.

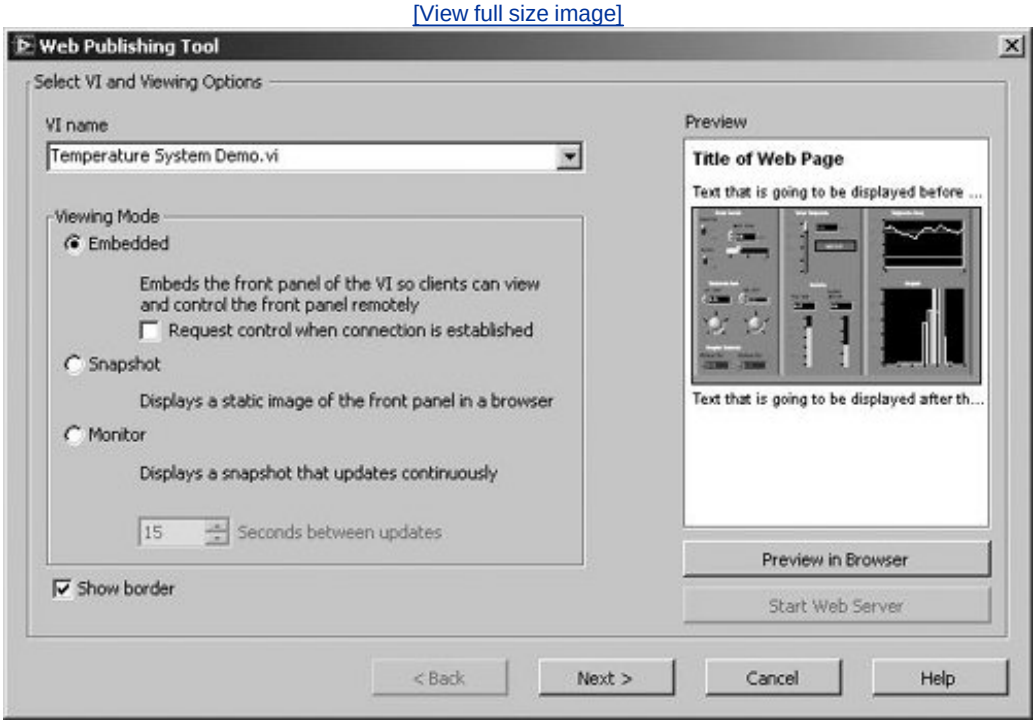
2. Open the VI Temperature System Demo.vi (from the LabVIEW examples, in `/examples/apps/tempsys.llb`).

3. Open the Web Publishing Tool (from the Tools menu).

4. Set the VI name drop-down list to "Temperature System Demo.vi" and set the Viewing Mode option to "Embedded," as shown in Figure 16.4. Then press the Preview in Browser button to open a preview HTML page in the default web browser (you can also press the Next button to complete the steps necessary to permanently save the HTML file).

Figure 16.4. Web Publishing Tool dialog

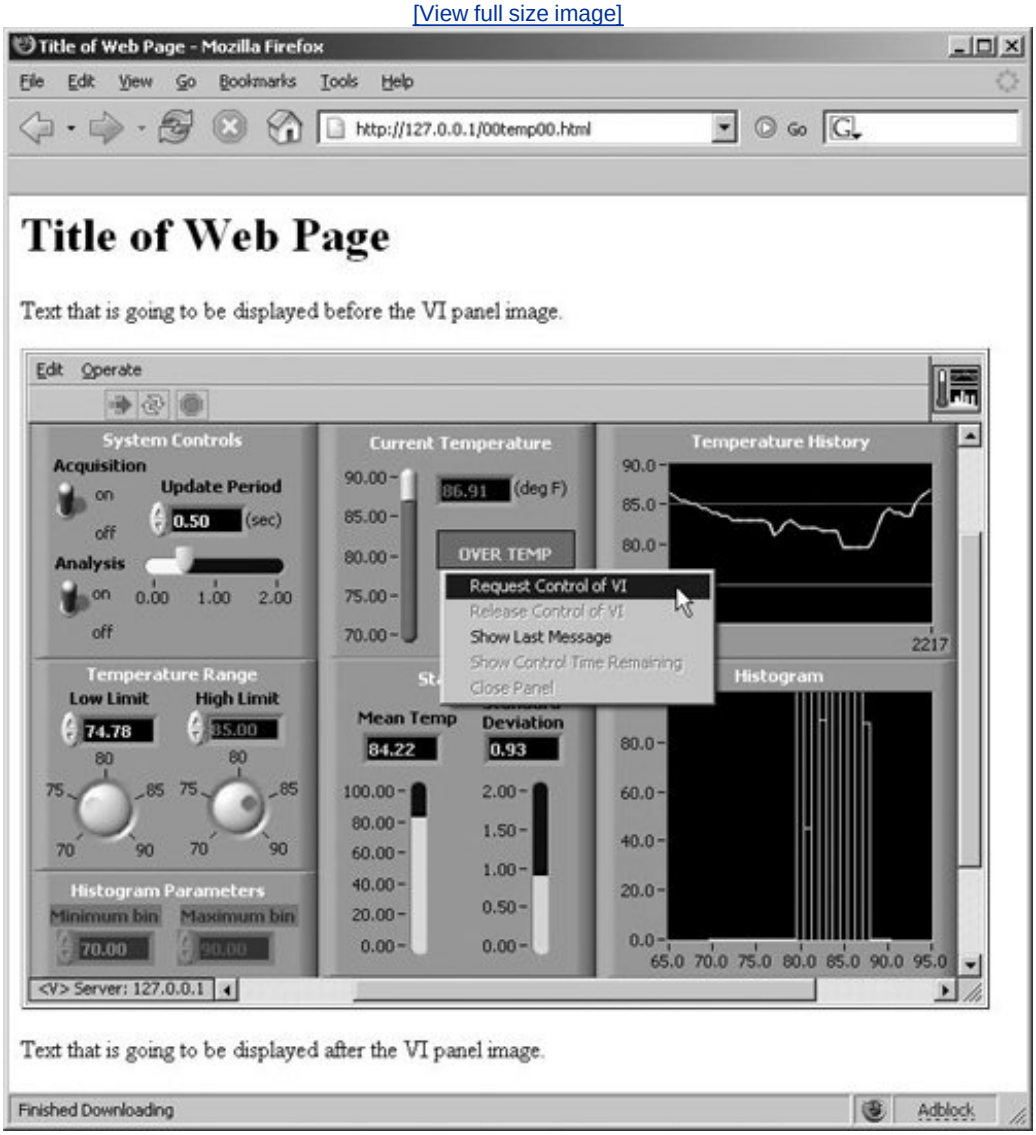
[View full size image]



5. In the page opened in your web browser, you will see the entire VI front panel (including the toolbar and menubar, if the VI is configured to show these while running). Right click on the panel and select Request Control of VI, as shown in Figure 16.5.

Figure 16.5. Requesting control of the VI in the web browser

[View full size image]



6. If control is granted (based on the Web Server: Browser Access settings in the LabVIEW Options . . . dialog), you will be able to control the VI, just as if you were running it from LabVIEW! Literally, all the functionality is theretry changing the scale values, or pop up on the graph and see the shortcut menu optionsit's all there!

7. You can release control, from the panel's pop-up menu via the Remote Panel Client>>Release Control of VI option. Note that only one remote panel can have control, at any given time.

8. You can regain control of the VI front panel (the real VI's front panel), from the pop-up menu of Temperature System Demo.vi, by selecting the Regain Control option. Additionally, you can also lock out other panels from gaining control by choosing the Remote Panel Server>>Lock Control option.

In addition to the built-in web server in LabVIEW, you can purchase the Internet Toolkit for LabVIEW from National Instruments. This toolset gives you an enhanced web server, with all the same features as the built-in web server, plus support for secure directories (user/password/groups), CGI capability, ftp, and telnet VIs. The following table shows the features that come with LabVIEW 8.0 and those that are part of the Internet Toolkit add-on:

LabVIEW 8.0	*+ Internet Toolkit*
http server (Web) server	telnet client
smtp client (email)	http server with added capabilities, such as CGI and password-protected directories
Remote panels	http client

PREV NEXT

Emailing Data from LabVIEW

In some applications, it's useful to program your LabVIEW application to notify someone by email or send data by email programmatically. LabVIEW has several VIs for sending email on the Data Communication>>Protocols>>SMTP Email palette. These VIs have options for sending text-only email, email with attachments, and email to multiple recipients (see Figure 16.6).

Figure 16.6. SMTP Email palette

These VIs require that you specify the Simple Mail Transfer Protocol (SMTP) server's hostname or IP address that will be used for sending emailthe same type of settings that your desktop email application uses. You must be able to communicate with this server in order to send email. Also, these VIs do not support SMTP servers that require password authentication. You might need to talk to your company's Information Technology (IT) department or your Internet Service Provider (ISP) in order to learn your SMTP server address and settingsthey can hopefully help you find a workable solution. Finally, if you do need support for password authentication with your SMTP server, visit OpenG.orgthere you can find some free SMTP VIs that support password authentication.

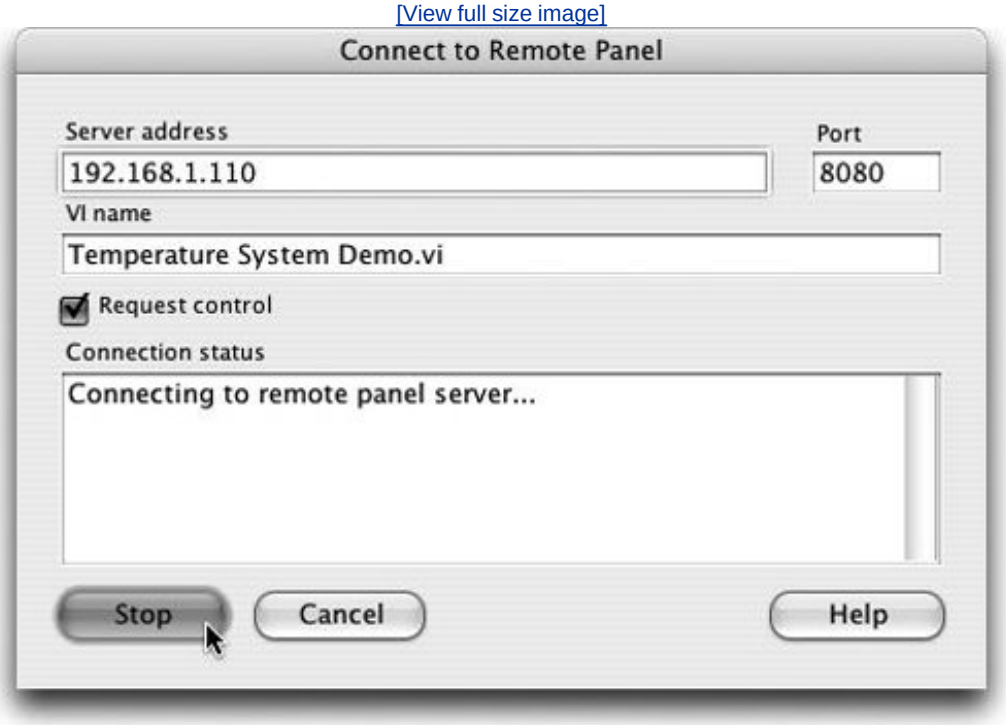
Remote Panels

Remote panels are almost identical in nature to the Embedded Panel option for web pages served up by LabVIEW's built-in web server (we learned about this option in the section,"Publishing to HTML with LabVIEW's Web Server"). The only difference is that remote panel clients connect from within LabVIEW and the remote panel appears as a separate VI window, rather than an embedded window in a web page. (The Web Server Embedded Panel option actually uses the remote panel technology under the hood.)

To connect to a remote panel from within LabVIEW, select Operate>>Connect to Remote Panel . . . from the menu. This will open a dialog where you can enter your connection parameters, and connect to the server, as shown in Figure 16.7.

Figure 16.7. The Connect to Remote Panel dialog

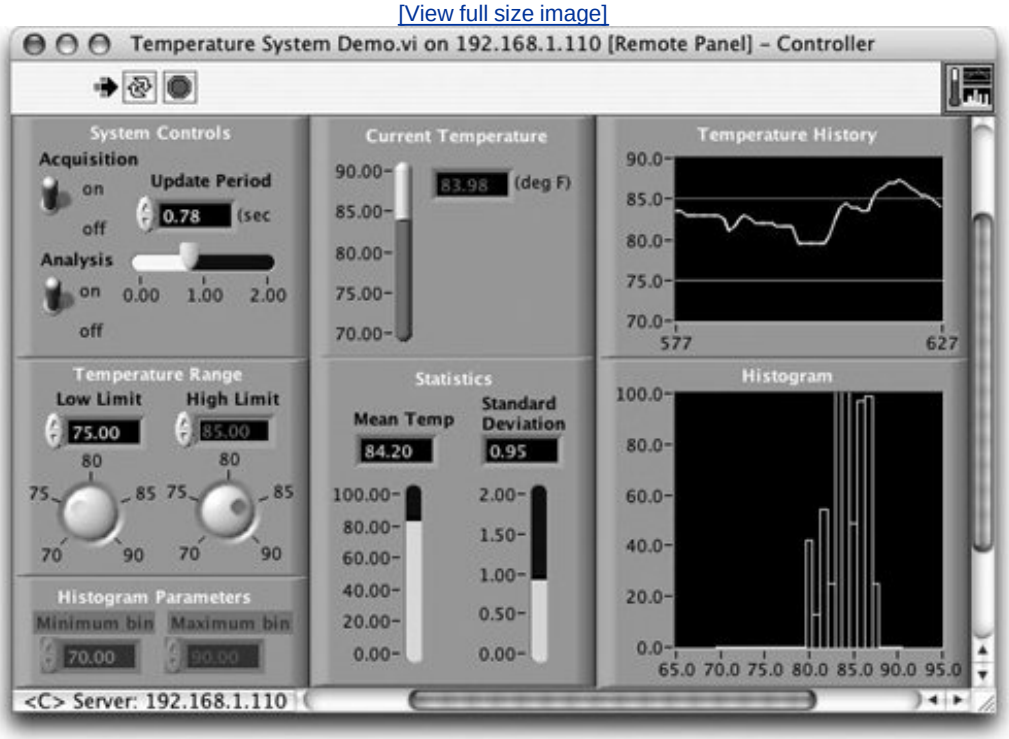
[View full size image]



Once you are connected, the remote panel client window (see Figure 16.8) will appear and you will be able to take control and use the VI, just as if it was running on your computerbut don't forget, the VI is actually running on the remote system.

Figure 16.8. Remote panel client window connected to a VI on a remote computer

[View full size image]



In order for remote panel (and Web Server Embedded Panel) clients to connect to your VIs, you will need one remote panel license per simultaneous client connection. The Base version of LabVIEW does not include any remote panel licenses, the Full version includes one license, and the Professional version includes five licenses. You can purchase additional licenses from National Instruments.

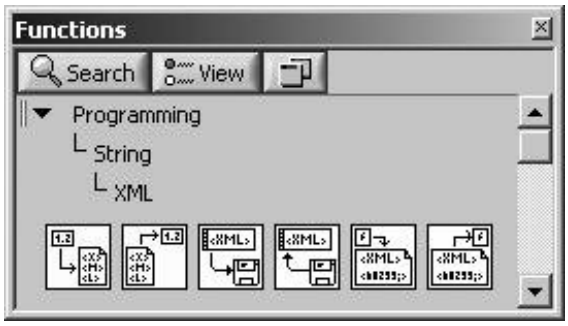
PREV NEXT

Self-Describing Data: XML

Chances are, you've probably heard of XML. If you use XML, you'll find this section very enlightening as to how LabVIEW can describe its data in XML. If you have never used XML, read on anyway, as you might find this helpful in the future.

XML, short for Extensible Markup Language, is a set of rules for formatting data as text. The XML VIs (found on the Programming>>String>>XML palette) allow you to work with XML data and files (see Figure 16.9).

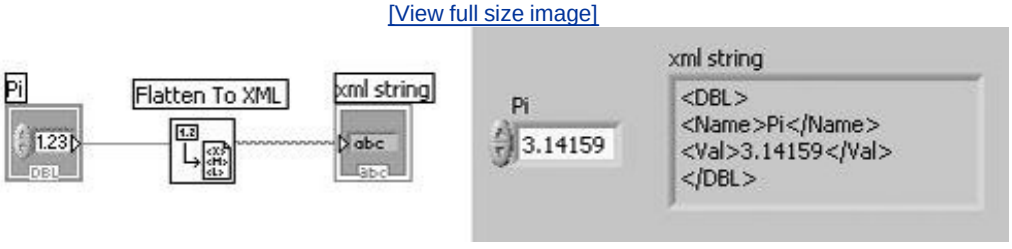
Figure 16.9. XML palette



XML was created for the purpose of allowing data to be shared across systems in a format that is independent of both programming language and platform. Figure 16.10 shows an example of how we can use the Flatten To XML function to convert a DBL numeric to an XML string.

Figure 16.10. Flatten To XML used to convert a LabVIEW numeric data into an XML representation

[View full size image]



As we can, see our DBL numeric is converted into the following text:

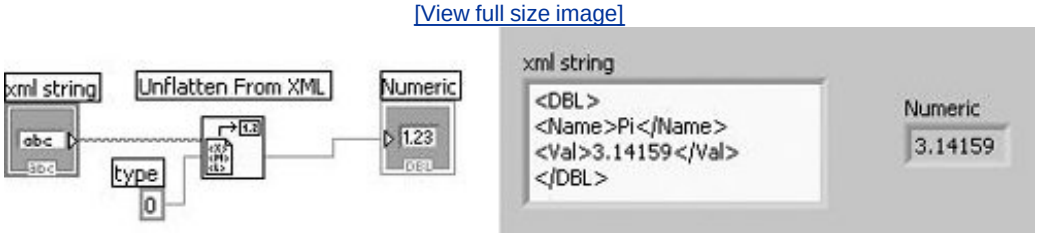
```
<DBL>
<Name>Pi</Name>
<Val>3.14159</Val>
</DBL>
```

The characters `<DBL>` signify the beginning of a data *entity* named "DBL" and the characters `</DBL>` signify its end. Everything in between is the data of the "DBL" entity. The "DBL" entity contains two sub-entities: "Name" (the data name) and "Val" (the data value). As you can see, the entities "Name" and "Value" do not have sub-entities; but rather, they have text that specifies the data values of the "Name" and "Value" entities.

We can convert our XML string back into LabVIEW data using the Unflatten From XML function, as shown in Figure 16.11. If either the format or structure of the XML data is not correct, Unflatten From Error will output an error.

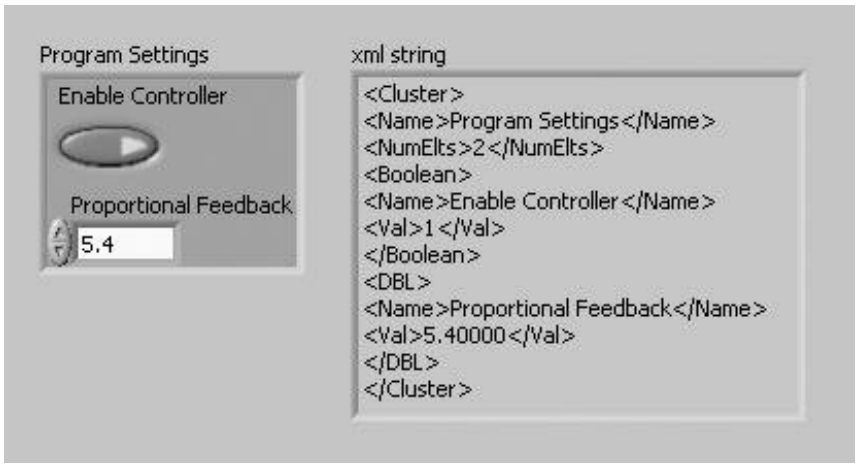
Figure 16.11. Unflatten From XML used to convert an XML representation of a numeric into a LabVIEW numeric

[View full size image]



Now, let's look at the flattened XML data of a more complex LabVIEW data structure: a cluster with two elements (see Figure 16.12).

Figure 16.12. A LabVIEW cluster (left) along side its XML representation (right)



As you can see, our `<Cluster>` has four sub-entities: `<Name>`, `<NumElts>`, `<Boolean>`, and `<DBL>`. Notice also that `<Boolean>` and `<DBL>` have sub-entities themselves. It might be a little difficult for you to see the nesting of the entities within the XML string that is output by Flatten To String. To make discerning the relationships easier, we can apply indentation to each entity based on its level of nesting:

```
<Cluster>
  <Name>Program Settings</Name>
  <NumElts>2</NumElts>
  <Boolean>
    <Name>Enable Controller</Name>
    <Val>1</Val>
  </Boolean>
  <DBL>
    <Name>Proportional Feedback</Name>
    <Val>5.40000</Val>
  </DBL>
</Cluster>
```

Applying indentation does not change the XML data, or entity relationships. In fact, XML ignores whitespace (tabs, spaces, carriage returns, line feeds, etc.) in between entities. However, XML *does not* ignore white space in an entity that is used for storing text data, such as the <Name> and <Value> entities.

LabVIEW Data XML Schema

XML does not specify anything about clusters, Booleans, DBLs, or any of LabVIEW's data types. However, XML does specify the rules for parsing entity tags and sub-entities. In order to validate that an XML data structure *is* LabVIEW data, we need to validate it against a set of rules, called a *schema*, that define the allowed types of (and relationships between) entities, sub-entities, and textual data.

The LabVIEW XML schema is stored in the following file:

`vi.lib\Utility\LVXMLSchema.xsd`

The file extension `.xsd` stands for XML Schema Definition, and the file contains an XML schema. This file is itself an XML file that conforms to an XML schemarecursion and self-consistency in a model is very, very cool!

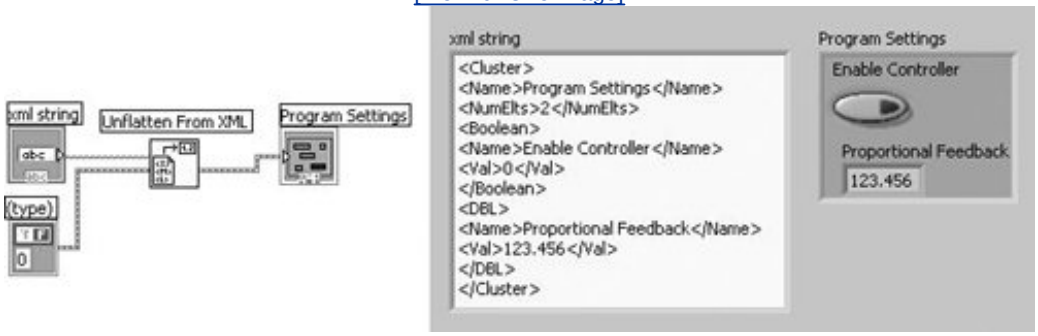
We aren't going to go into the details of the XML Schema Definition format here, but you should remember where the LabVIEW XML schema file (`LVXMLSchema.xsd`) is located. If you are sending LabVIEW XML data to someone, they will probably need it in order to validate the integrity of the data (especially if they are not using LabVIEW).

XML Parser

If XML data conforms to the LabVIEW XML schema (for example, if it was created using the Flatten To XML function), then you can use the Unflatten From XML function to convert the data back into LabVIEW data (shown in Figure 16.13).

Figure 16.13. Using Unflatten From XML to parse XML data that conforms to the LabVIEW XML schema

[View full size image]



The Flatten To XML function represents floating point numerics using the %.6 format (up to six decimal places). If you need to store floating point numeric data with more digits of precision, then you should use the Flatten To XML VI that is provided in the OpenG

Toolkitit allows you to specify any format string that you wish. Additionally, LabVIEW's Flatten To String and Unflatten From String functions do not handle variant data; but the OpenG Toolkit Flatten To XML and Unflatten From XML VIs also overcome this limitation. See Appendix C, "Open Source Tools for LabVIEW: OpenG," for more information.

There are other XML parsers that you can call from LabVIEW, which allow parsing data of *any* XML schema and validating the data against that schema. You can use the Microsoft XML Parser, which is accessible via ActiveX and .NET. You can also use the XML VIs offered in the Internet Toolkit from National Instruments. Another option is an open source tool called LabXML, which is available from http://labxml.sourceforge.net/.

PREV NEXT

Sharing Data over the Network: Shared Variables

In applications where you have several computers that need to read or write from some common data variables, LabVIEW provides you with an elegant solution (that is, for the most part, only supported on Windows machines): shared variables. (On other operating systems, the older DataSocket technology provides a similar but far more limited capability.) With shared variables, VIs on different machines on a network can read from or write to these variables, without your having to worry about how to program or handle the communication.

Shared variables also allow you to send data between VIs located in the same application, not just across a network. They allow you to specify options for buffering and single-writer restriction. As you can see, this is a very powerful feature that makes it very easy to create distributed networked applications.

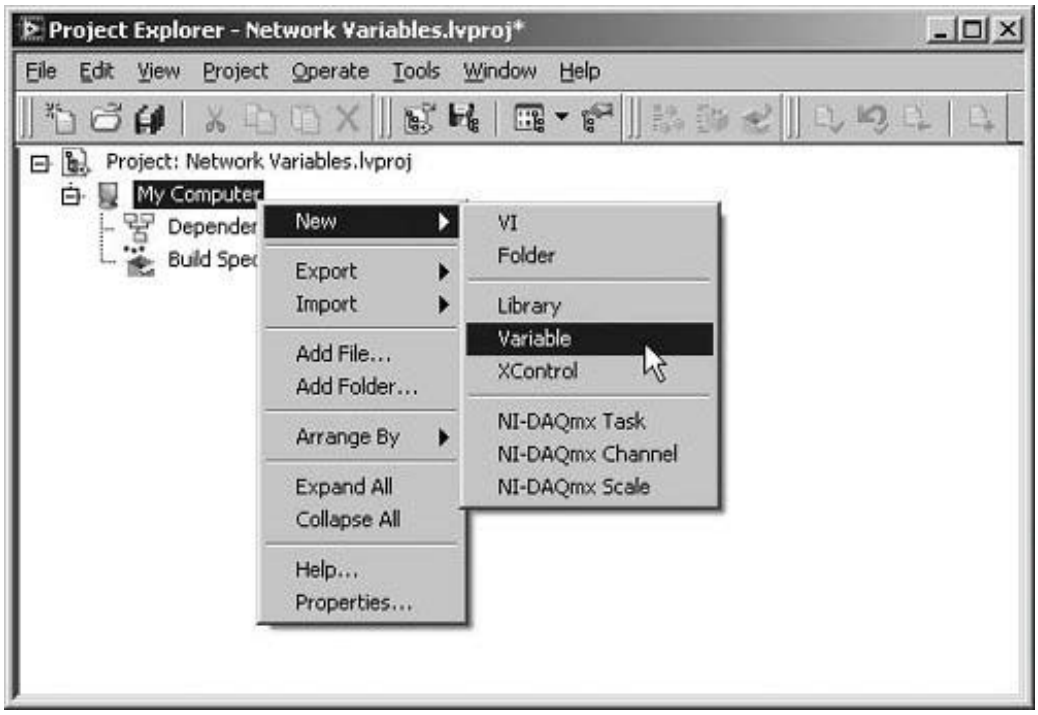
Shared Variables in LabVIEW Projects

To create a shared variable, create a new LabVIEW project (from the File menu, select New . . ., and select Empty Project from the ensuing dialog).

Then, in your LabVIEW Project, pop up on My Computer and select New>> Variable, as shown in Figure 16.14.

Figure 16.14. Creating a new variable in a LabVIEW project

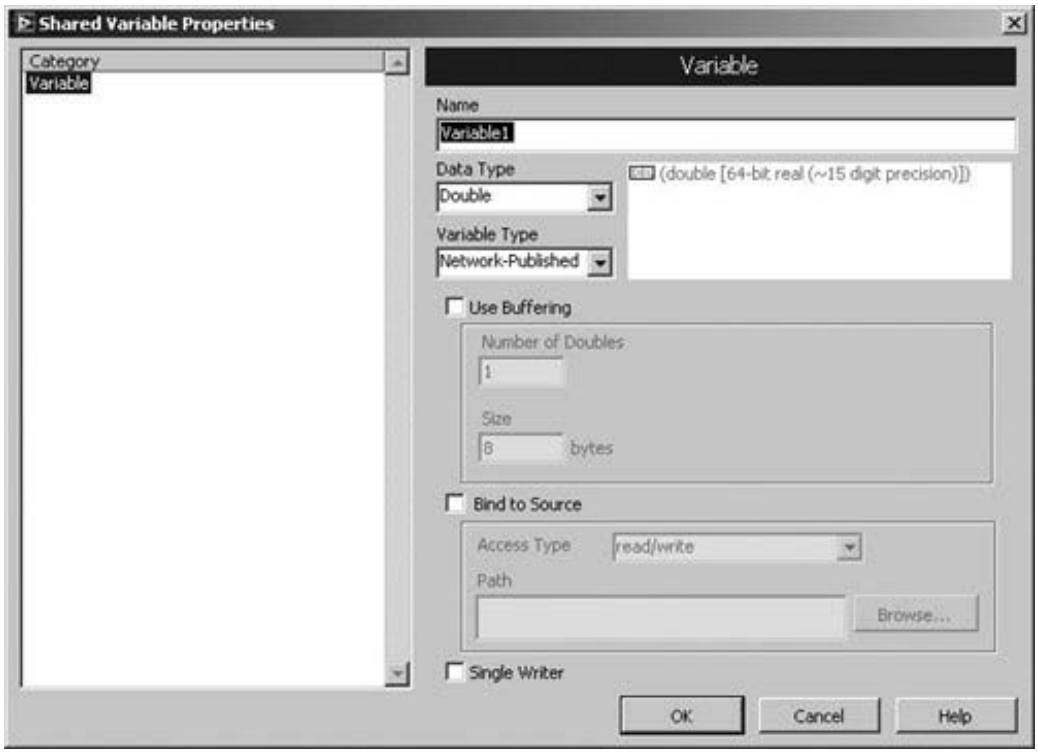
[View full size image]



You will be presented with the Shared Variable Properties dialog (see Figure 16.15).

Figure 16.15. Shared Variable Properties dialog

[View full size image]

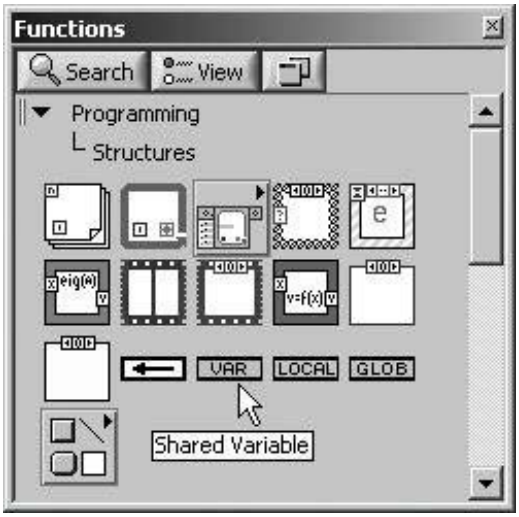


The Shared Variable Properties dialog allows you to specify the following options for your shared variable:

- Name Specify the name of your shared variable.
- Data Type Specify the data type of your shared variable.
- Variable Type Whether your shared variable is a Network-Published variable that can be accessed from remote computers and targets, or a Single Process variable that can only be read on the local computer. LabVIEW modules and toolkits you have installed might provide additional shared variable types, configuration options, and limitationssee the documentation for the specific module or toolkit for more information. (For example, the LabVIEW Real-Time Module installs the Time-triggered variable, which provides deterministic communication over Ethernet networks.)
- Buffering Enable data buffering and configure the buffer size.
- Bind to Source Configure the data source of the shared variable. You can choose from DAQ channels, other shared variables, and data items from I/O servers outside the active project.
- Single Writer This option specifies whether to allow writing to the variable from only one location. If you choose this option, all other locations can only read from the shared variable.

Once you have configured your shared variable settings, you can read or write to it from within your LabVIEW VIs using the Shared Variable structure (found on the Programming>>Structures palette), shown in Figure 16.16.

Figure 16.16. Shared Variable structure shown on the Programming>>Structures palette

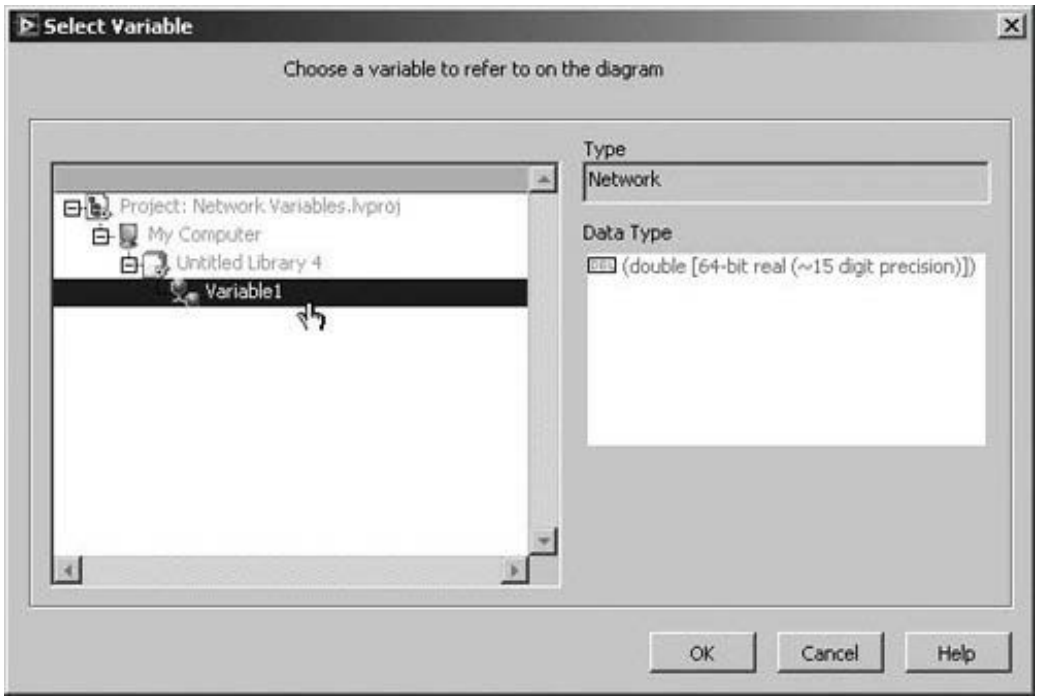


Shared Variable Structure (Unassigned)

When first placed on the block diagram, the Shared Variable structure will not be assigned to any shared variable. Double-click the Shared Variable structure (or choose Select Variable from its pop-up menu) to open the Select Variable dialog, shown in Figure 16.17.

Figure 16.17. Using the Select Variable dialog to assign a variable to a Shared Variable structure

[View full size image]



Shared Variable Structure (Assigned)

Use this dialog to select a shared variable. After you select the desired variable from the tree control and press OK, the Shared Variable structure will be assigned to the selected variable and change its appearance to reflect the color of the variable's data type and the name of the variable.

You can configure a Shared Variable structure as a Read Shared Variable or a Write Shared Variable from its pop-up menu using the Change to read and Change to write options, respectively. Figures 16.18 and 16.19 show the differences in appearance between the two. Notice that the Read Shared Variable has a thicker border (because it is a data source) and has an arrow showing that data flows out from its right side. Likewise, the Write Shared Variable has a thinner border (because it is a data sink) and has an arrow showing that data flows into it on its left side.

Figure 16.18. Read Shared Variable

Figure 16.19. Write Shared Variable

The preceding instructions showed you how to assign a new (empty) Shared Variable structure to a shared variable in your LabVIEW project. However, an easier way to achieve the same result is to drag and drop a shared variable from the Project Explorer window onto the block diagram of any VI. This will create a Shared Variable structure on the block diagram that is associated with the one that was dragged from the Project Explorer window.

In some ways, shared variables are a lot like global variables; whereas globals share data within a set of VIs on a single computer, shared variables can share data across multiple computers on a network.

Unlike global variables, shared variables have special synchronization capabilities built into the model that can help you avoid the type of race conditions that often occur from improper use of regular globals. Because of these intricacies, you will definitely want to take the time to understand how they work, before using them casually. For considerations and detailed instructions related to deploying applications using shared variables, you will want to refer to the Fundamentals>>Networking in LabVIEW>>Concepts>>Sharing Live Data Using Shared Variables section of the LabVIEW Help.

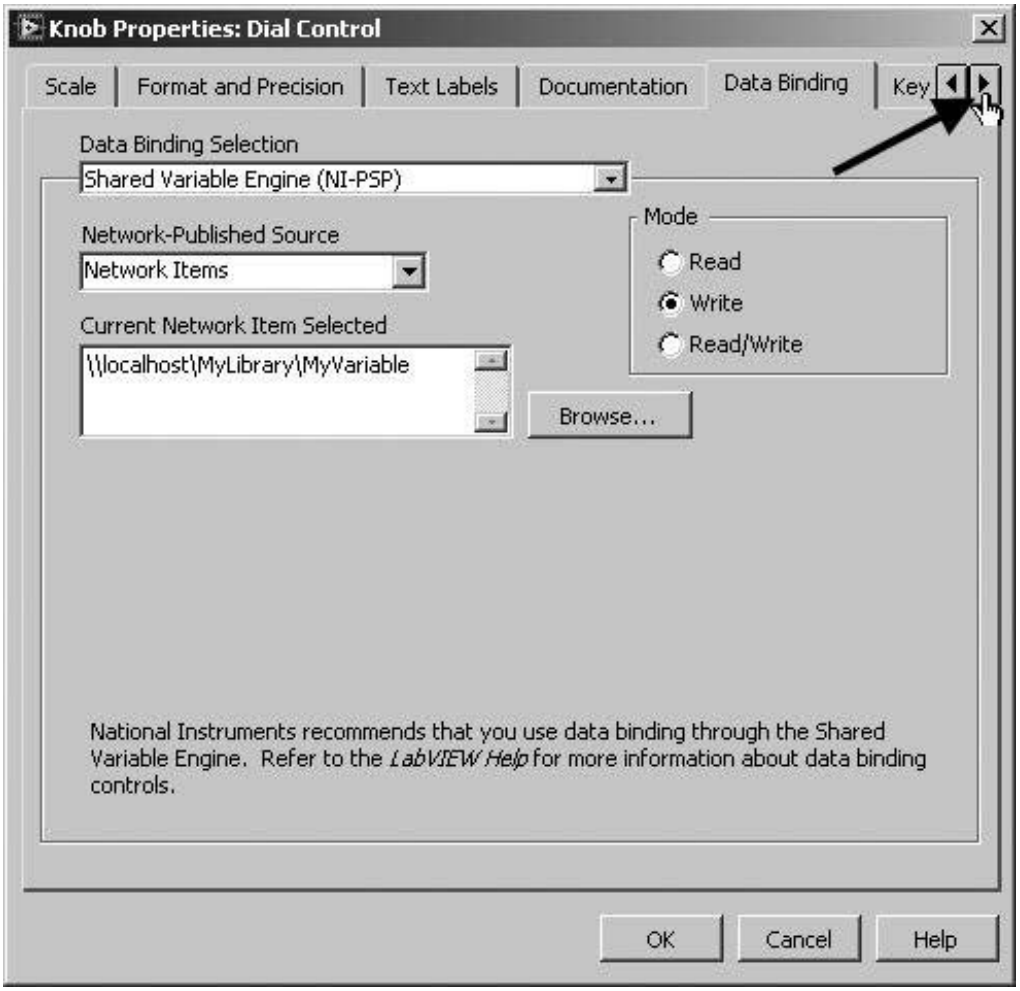
Binding Controls and Indicators to Shared Variables

A very simple way to write and read to shared variablesit requires no programming at allis *data binding*. Data binding allows a control or indicator to be connected, behind the scenes, to a shared variable. When a user changes the value of a control that is bound to a shared variable, the value is written to the shared variable. When a shared variable's value changes, any indicator that is bound to the shared variable will update its value to that of the shared variable. It's that easy, and it doesn't require any code on your block diagram!

To bind a control or indicator to a shared variable, first open its Properties dialog by right-clicking on the control or indicator and selecting Properties . . . from the pop-up menu. Browse to the Data Binding tab of the Properties dialog. You may have to press the *right-arrow* button in the upper-right corner of the tab control to make the Data Binding page visible (this right-arrow button is highlighted in Figure 16.20).

Figure 16.20. Use the right-arrow button to find the Data Binding tab of

the Properties dialog, where you can configure the binding of a control to a shared variable.



Set Data Binding Selection to Shared Variable Engine (NI-PSP), set Network-Published Source to Network Items, and then press the Browse button to select the desired shared variable from the network.

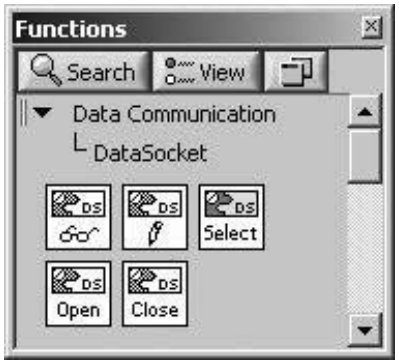
Finally, choose the desired Mode from the following options:

- Read Updates a control or indicator value whenever the value of the shared variable changes
- Write Write to the shared variable whenever the user changes the value of the control.
- Read / Write Allows both the Read and Write mode behaviors. The control will update when the shared variable value changes, and the control will update to reflect changes in the value of the shared variable.

Programmatically Accessing Shared Variables Using DataSocket

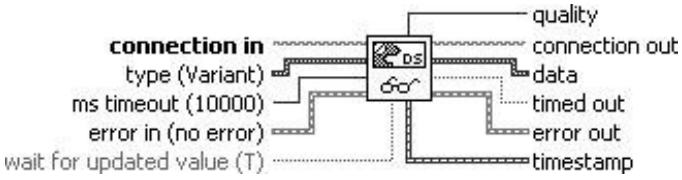
You can read and write to shared variables programmatically using the DataSocket Read and DataSocket Write functions, which can be found on the Data Communication>>DataSocket palette (see Figure 16.21).

Figure 16.21. DataSocket palette



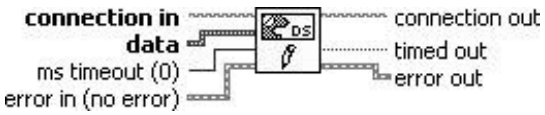
DataSocket Read (Data Communication>>DataSocket palette) dequeues the next available data value from the client-side buffer associated with the connection you specify in connection in and returns the data (see Figure 16.22).

Figure 16.22. DataSocket Read



DataSocket Write (Data Communication>>DataSocket palette) writes data to the connection you specify in connection in. The connector pane displays the default data types for this polymorphic function (see Figure 16.23).

Figure 16.23. DataSocket Write



Shared Variable URLs

In order to use the DataSocket VIs to read or write to shared variables, you will need to know the computer name, project library name, and shared variable name where the shared variable resides.

Shared variables must be created inside LabVIEW Project Libraries. You will need to know the library name, in order to access it programmatically using the DataSocket functions.

With this information, you will construct a URL (which stands for Uniform Resource Locator) string that has the following format:

```
psp://computer/library/shared_variable
```

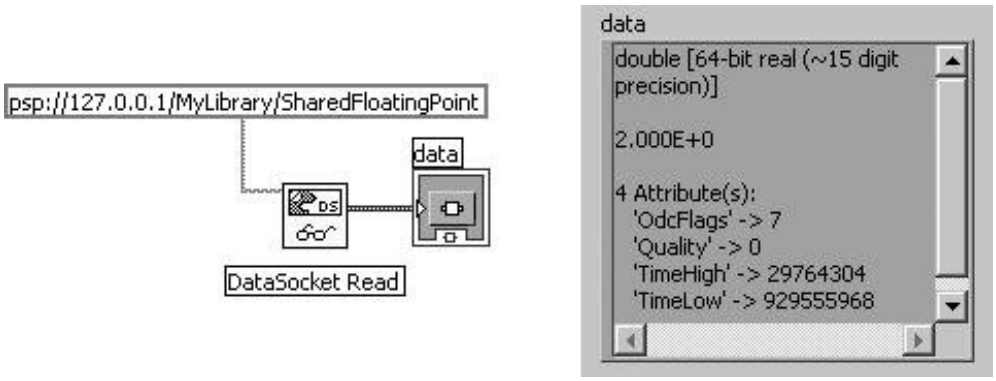
Or, if your shared variable resides in a sublibrary of a library, then the URL will have the following format:

```
psp://computer/library/sub_library/shared_variable
```

Reading Shared Variables Programmatically

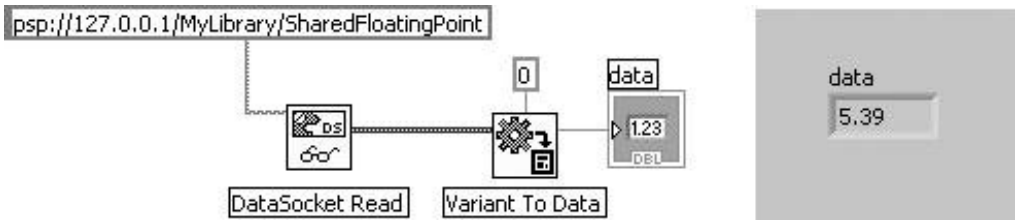
To read a shared variable's value, you will pass the shared variable's URL to the DataSocket Read function, as shown in Figure 16.24.

Figure 16.24. Using DataSocket Read to read a shared variable's value programmatically



The data returned from the DataSocket Read function is the shared variable's value as a variant data type. This is where all of the skills you learned in Chapter 14, "Advanced LabVIEW Data Concepts," for operating on variants will come in handy. Use the Variant To Data function, as shown in Figure 16.25, to convert the variant into a DBL numeric.

Figure 16.25. Converting the variant returned by DataSocket Read to a numeric type

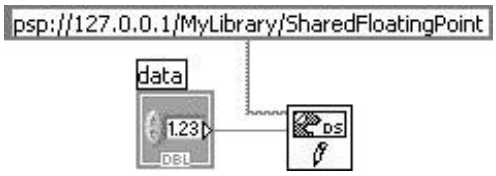


You probably also noticed in Figure 16.25 that there are attribute data inside the variant that is returned by DataSocket Read. These attributes can be accessed using the Get Variant Attribute function that you learned about in Chapter 14.

Writing Shared Variables Programmatically

In order to write to a shared variable's value programmatically, you will use the DataSocket Write function, passing it the value to write and the shared variable's URL.

Figure 16.26. Using DataSocket Write to write a shared variable's value programmatically

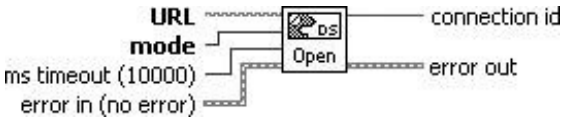


Maintaining Shared Variable Connections

When you wire a URL into the DataSocket Read and DataSocket Write functions, they establish a network connection, perform the read or write operation, and then disconnect. If you are going to read or write many times, you may wish to stay connected in order to avoid spending time establishing the connection each time you want to read or write the data. In this case, you will want to use the DataSocket Open function to obtain a connection reference (connection id) that can be passed to DataSocket Read and DataSocket Write. When you are finished reading and writing data, use DataSocket Close to close the network connection.

DataSocket Open (Data Communication>>DataSocket palette) opens a data connection you specify in URL (see Figure 16.27).

Figure 16.27. DataSocket Open



DataSocket Close (Data Communication>>DataSocket palette) closes a data connection you specify in connection id (see Figure 16.28).

Figure 16.28. DataSocket Close

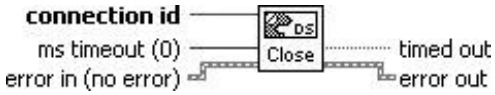
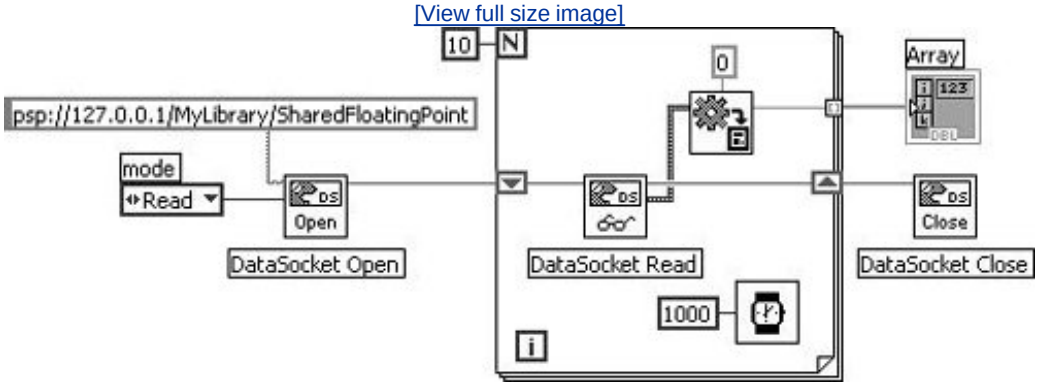


Figure 16.29 shows an example of how to open a persistent connection to a shared variable and read its value multiple times before finally closing the connection.

Figure 16.29. Opening a persistent connection to a shared variable and reading its value multiple times while connected

[View full size image]



DataSocket Read and DataSocket Write allow you to wire either a URL or a connection id reference that was created using the DataSocket Open function. If you wire a URL, then they will connect, read or write, and then disconnect. If you wire a connection id, then they will use that connection and leave the connection open after they are finished.

PREV NEXT

Talking to Other Programs and Objects

Whether you are on Mac OS X, Linux, or Windows, LabVIEW has tools that will help you communicate with other applications and objects on those systems. On Windows, there is support for .Net and ActiveX; on Mac OS X, there is support for Apple Events and pipes; and on Linux, there is support for pipes. We will now learn all about these exciting technologies!

.NET and ActiveX

.NET and ActiveX are frameworks from Microsoft that allow Windows applications to communicate with one another and to embed user interface components from one application into another. For example, with ActiveX, you can embed a Microsoft Excel spreadsheet into a Microsoft Word document (this used to be called OLE).

In order to use the .NET functions and controls in LabVIEW, you will need to have .NET Framework 1.1 Service Pack 1 or later installed. You can find out more information about these requirements in the LabVIEW Release Notes. You can download the .NET Framework from Microsoftit should be easy to find, using your favorite search engine or by searching the microsoft.com web site.

You should, however, be able to use the ActiveX functions and controls without installing any additional software.

.NET and ActiveX are very broad topics, and we'll only point you to a few things LabVIEW can do with them. Needless to say, only LabVIEW for Windows supports .NET and ActiveX, because they are Windows protocols.

The .NET and ActiveX technologies supported by LabVIEW are grouped into two main categories: Servers and Controls. A server is an application that runs outside of LabVIEW, and a control is a component that we will place on the front panel of a LabVIEW VI (for example, a LabVIEW front panel can contain a Microsoft Excel worksheet).

The .NET and ActiveX functions palettes (shown in Figures 16.30 and 16.31) are found beneath the Connectivity category of the Functions palette.

Figure 16.30. .NET palette

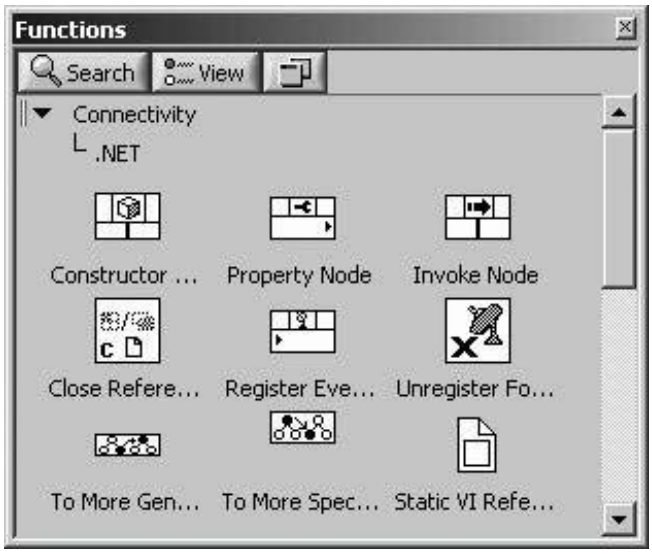
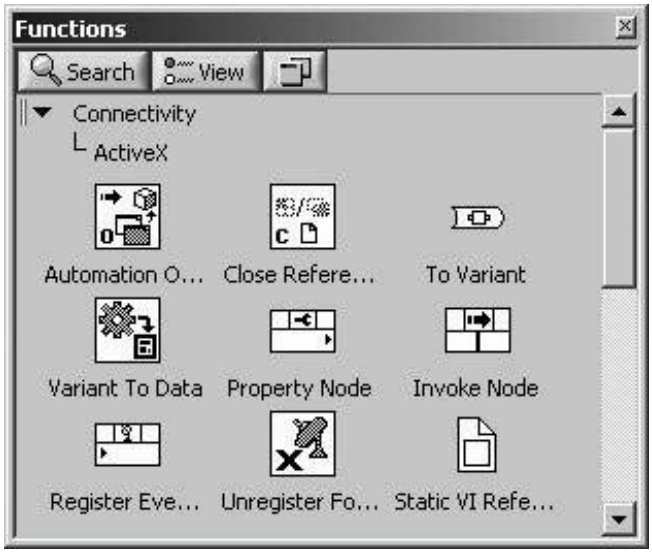


Figure 16.31. ActiveX palette



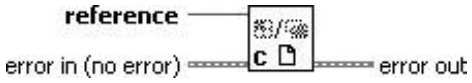
Communicating with .NET and ActiveX Servers

The functions in the following table allow you to open and close references to .NET and ActiveX servers.

(.NET) Constructor Node (Connectivity>>.NET palette) creates an instance of a .NET object.

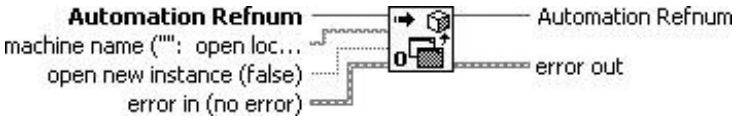
Double-click this node (or choose Select Constructor from its pop-up menu) to display the Select .NET Constructor dialog box and choose a .NET Assembly (see Figure 16.32).

Figure 16.32. (.NET) Constructor Node



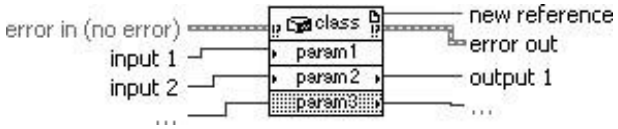
(ActiveX) Automation Open (Connectivity>>ActiveX palette) opens an automation refnum that points to a specific ActiveX object. Select the ActiveX class by wiring an automation refnum constant or control to the Automation Refnum input. The refnum can be configured by choosing Select ActiveX Class>>Browse from its pop-up menu (see Figure 16.33).

Figure 16.33. (ActiveX) Automation Open



The Close Reference function closes .NET and ActiveX server references opened with the preceding functions. It may be found on the Connectivity>>ActiveX, Connectivity>>.NET, and Programming>>Application Control palettes (see Figure 16.34).

Figure 16.34. Close Reference



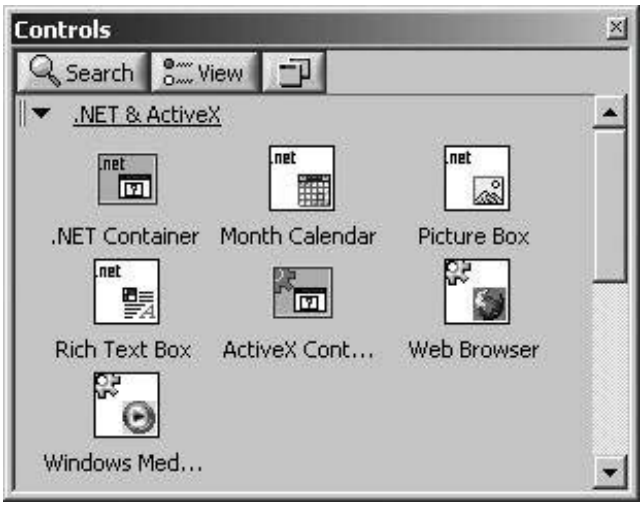
Once you have obtained a reference to a .NET or ActiveX server using the aforementioned functions, you can use the Property Node, Invoke Node, and Register Event Callback functions to read and write properties, call methods, and register for events (respectively).

Embedding .NET and ActiveX Controls on the Front Panel

One very useful way to use .NET and ActiveX in LabVIEW is to take advantage of the .NET Container and ActiveX Container (found on the .NET & ActiveX subpalette of the Controls

palette, shown in Figure 16.35). These allow you to embed objects or documents from one application into another.

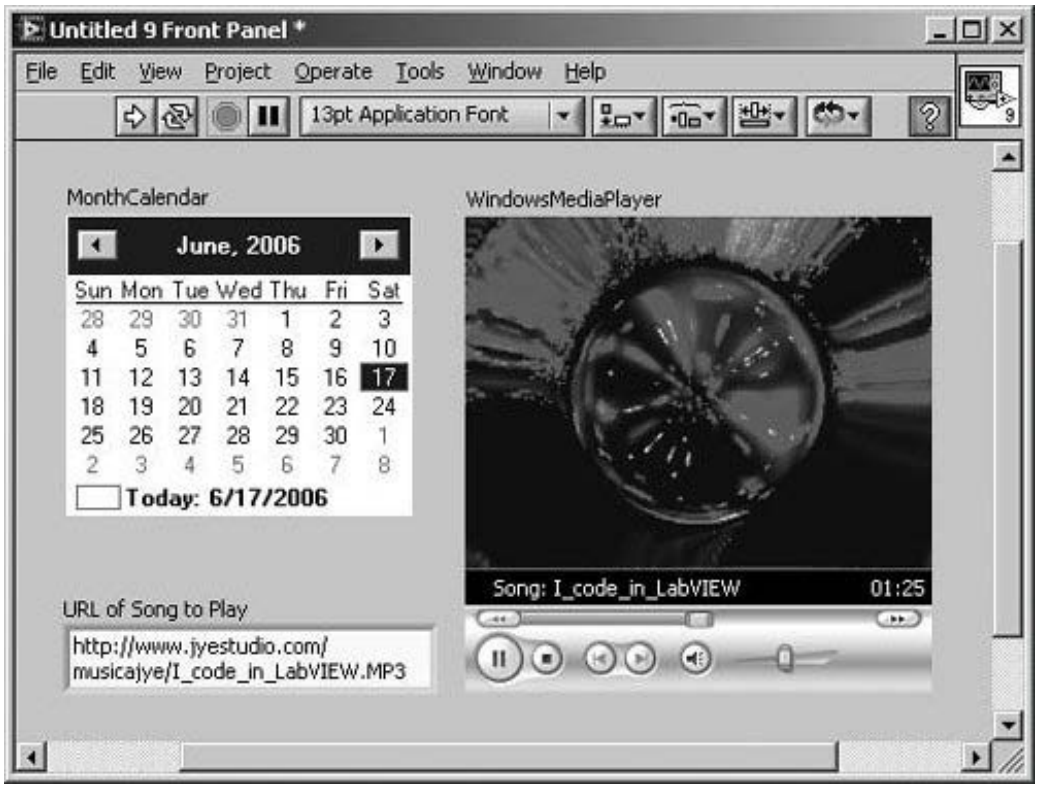
Figure 16.35. The .NET & ActiveX palette



The .NET & ActiveX palette also includes several containers that have been preconfigured for certain controls such as the Month Calendar, Web Browser, and Windows Media Player. Figure 16.36 shows a front panel where we have placed Month Calendar and Windows Media Player controls from the .NET & ActiveX palette.

Figure 16.36. .NET & ActiveX controls on a VI front panel

[View full size image]



Once you have placed a .NET Container or ActiveX Container onto the front panel, you can insert a control by selecting Insert .NET Control or Insert ActiveX Object (respectively) from its pop-up menu. This will present you with a dialog for the desired control.

On the block diagram, the .NET Container and ActiveX Container terminals output a reference that can be wired to the Property Node, Invoke Node, and Register Event Callback functions to read and write properties, call methods, and register for events (respectively).

You do not need to explicitly open or close references to .NET or ActiveX controls. Just use the reference that flows out of the .NET Container or ActiveX Container terminal on the block diagram.

Do the following activity to see how you can embed a web browser on your front panel.

Activity 16-3: Embedding an ActiveX Web Browser in a VI (Windows Only)

In this activity you will embed a web browser ActiveX control in a VI's front panel.

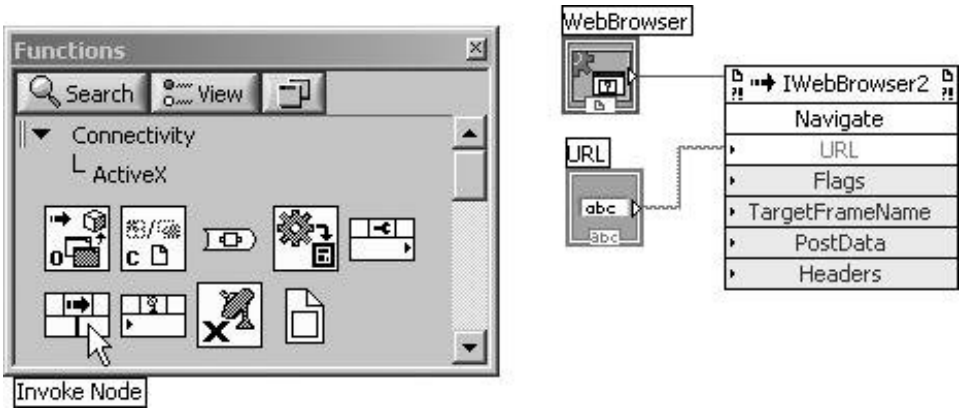
1. Open a new VI in LabVIEW. From the Controls>>.NET & ActiveX palette, select the ActiveX Container object (see Figure 16.37). Resize it to be very large on your front panel. Do *not* select the Web Browser object. This activity is going to demonstrate how to configure the ActiveX Container object yourself, rather than using a preconfigured ActiveX Container.

Figure 16.37. ActiveX Container control found on the .NET & ActiveX palette

2. Pop up in the container and select Insert ActiveX object. From the Select ActiveX Object dialog box, select Create Control. A list of all available objects is displayed. Next, scroll through the list, select the Microsoft Web Browser control, and press the "OK" button.

3. Switch to the block diagram. You can now use automation functions with this refnum. Use the Invoke Node function from the Communications>> ActiveX palette. Once you wire the automation reference of the control container to the Invoke Node function, it will display the automation object (IwebBrowser2) it is accessing. You now can select from a list of methods it exposes by popping up on the Method terminal. Select the "Navigate" method (see Figure 16.38).

Figure 16.38. An Invoke Node shown on the ActiveX palette and on the block diagram configured to call the Navigate method of the IwebBrowser2 automation object

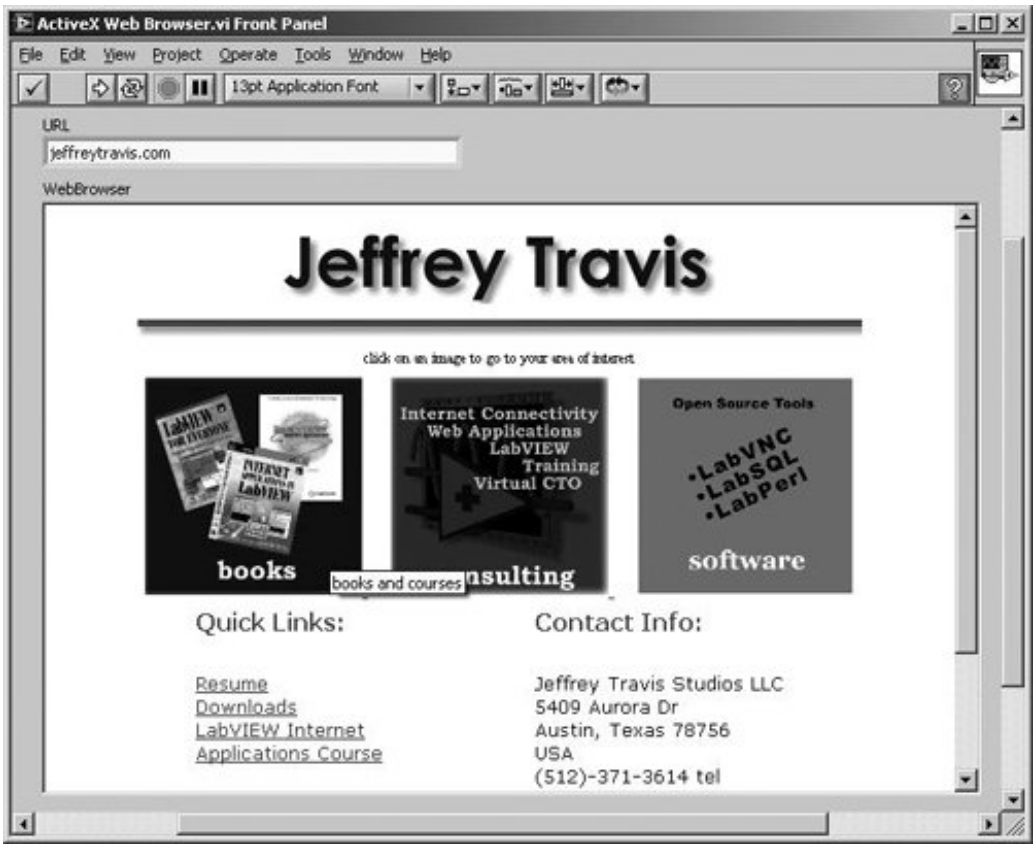


4. Pop up on the URL input and select Create Control. This will give you a string input for typing the URL you want to connect to.

5. Return to the front panel, type in a URL, and run the VI. You will see the web page appear in the container. In Figure 16.39, we typed `http://jeffreytravis.com`.

Figure 16.39. Your VI's front panel after you type in a URL and run the VI

[View full size image]



Unless you already understand ActiveX, this example may seem like a little magic. How is it possible to have a web browser as a LabVIEW front panel control? The answer is that the web browser is not really part of LabVIEW; in fact, LabVIEW is doing very little. Instead, the fact that both LabVIEW and Microsoft Internet Explorer (IE) can communicate via ActiveX allows you to embed the IE browser into LabVIEW.

Note several things about using this ActiveX web browser control:

- It is very easy to use and set up with minimal block diagram programming.
- You have access to many other web browser properties and methods, such as resizing the browser, moving forward and backward, refreshing the web screen, and so on.
- You can easily and quickly access both static HTML documents (via the file:// URL) and Internet web sites (via the http:// URL) from LabVIEW.
- This will only work on LabVIEW for Windows; it is not available for other platforms

This type of example is great for an application where you wish to link to an online web-based help system, for example.

You'll notice that the .NET and ActiveX functions are very similar to the VI Server functions; so if you understand the VI Server, you should have no problem using the .NET and ActiveX functions. The main difference is that you will need to understand the methods and properties of the external .NET

or ActiveX object (because LabVIEW can't know how it works). Many .NET and ActiveX components include a help file for the objects, which you can reference if it's available.

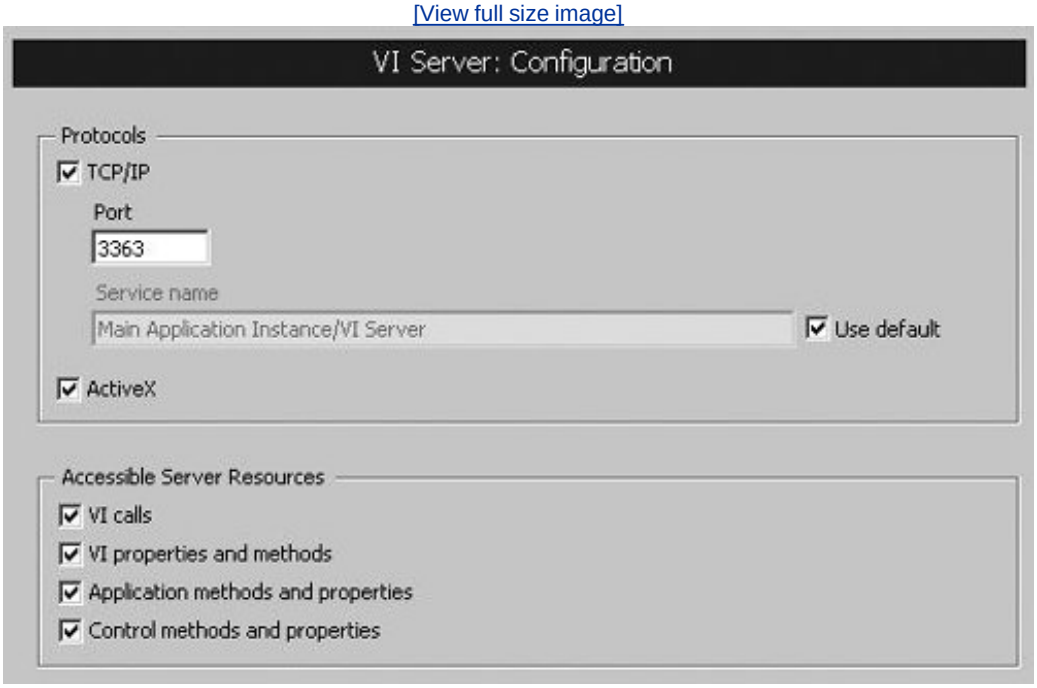
LabVIEW as an ActiveX Automation Server

LabVIEW can expose properties and methods of the LabVIEW application itself and of specific VIs to other ActiveX-enabled applications (for example, Microsoft Excel, Visual Basic, etc.). It does this through the VI Server interface. If you recall the VI Server discussion from Chapter 15, "Advanced LabVIEW Features," LabVIEW's VI Server capabilities are accessible by TCP/IP (other LabVIEW apps only), block diagram functions, and ActiveX.

To enable the ActiveX access to the VI Server in LabVIEW, you must check the appropriate box in Tools>>Options>>VI Server: Configuration, as shown in Figure 16.40.

Figure 16.40. VI Server configuration settings of the LabVIEW Options dialog

[View full size image]

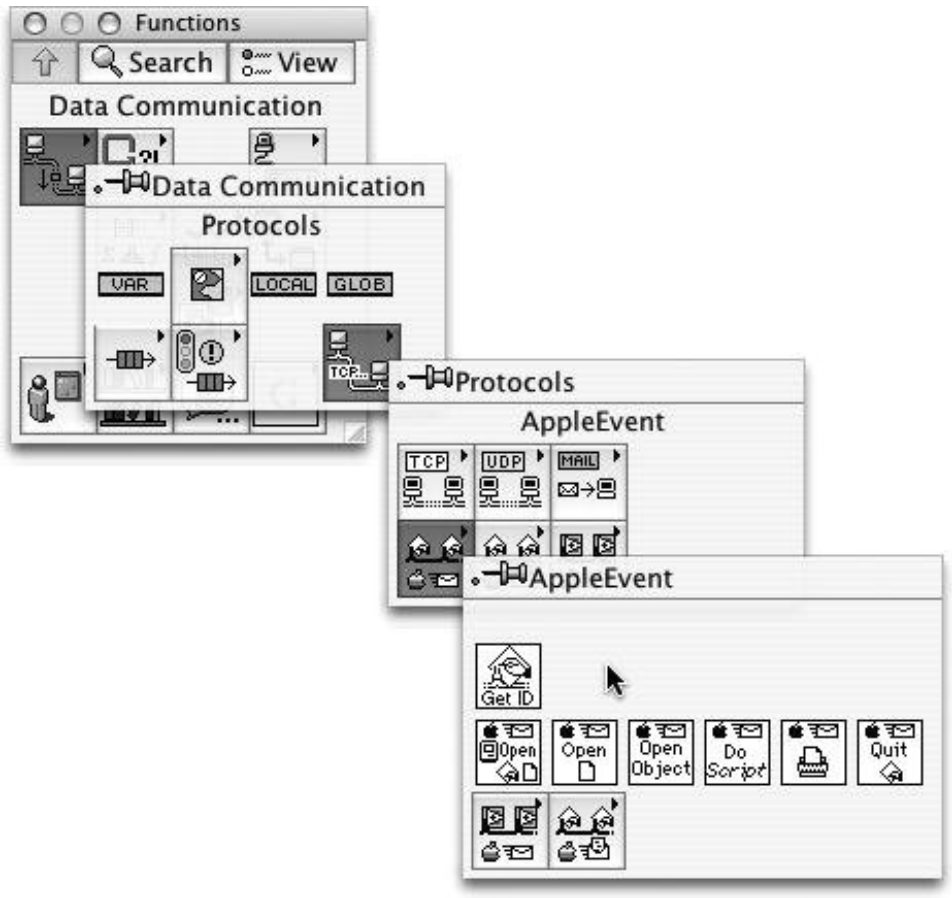


By enabling the ActiveX server, you can write external applications in other programming environments, such as Visual Basic or C++, which can interface to LabVIEW. For example, you could write a Visual Basic application that called a VI, showed its front panel, centered it on the screen, ran the VI, and closed itdoing all this programmatically through the ActiveX/VI Server interface. For more information on this capability, consult the LabVIEW manuals.

AppleEvents

On Mac OS X, you can use some LabVIEW VIs, found on the Data Communication>>Protocols>>AppleEvent palette, that provide interapplication communication through AppleEvents (see Figure 16.41).

Figure 16.41. AppleEvent palette



AppleEvents are a Mac OS-specific protocol that applications use to communicate with each other. AppleEvents send messages to other applications or to the operating system itself to open a document, request data, print, and so on. An application can send a message to itself, another application on the same computer, or another application on a remote computer.

You cannot use AppleEvent VIs to communicate with computers that aren't running Mac OS X (such as a PC running Windows). If you need to communicate with other platforms, use a protocol common to all of them, such as TCP/IP.

The actual low-level AppleEvent messages are quite intricateyou can use them with LabVIEW, but you'd better know the Mac OS very well and have a good reference handy. For simpler stuff, though, LabVIEW comes with higher-level VIs for sending some of the popular commands to applications: telling the Finder to open a document, for example. The AppleEvent VIs are located in the AppleEvent subpalette of the Communications palette.

You can find a neat example of how LabVIEW can communicate with Microsoft Excel in the full version of LabVIEW under `examples:comm:AE examples.llb`.

Pipes

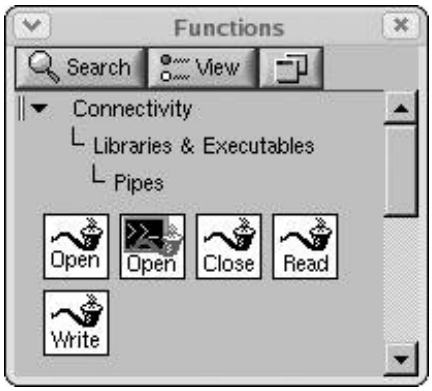
Pipes are a very cool tool, commonly used by computer hackers to transmit data between applications (by "hackers," we mean the good kind . . . like the IT guy in your company who wears a long beard, types 60 words per minute, has a stuffed penguin toy named "Tux" on his desk, and is always talking about how Windows isn't a *real* operating system). A pipe is basically just a First In First Out (FIFO) queue that acts as a conduit for data in inter-process communication (IPC). There are two types of pipes: *named* pipes and *command* pipes.

A named pipe is a special type of file. On Linux and Mac OS X, this can be created using the `mknod` and/or `mkfifo` commands (refer to the documentation for your OS shell). On Windows, named pipes are a little differentthey are not real files but exist beneath `\\.pipe\`, which is a purely virtual directory that Windows uses for named pipesand must be created using the Windows API.

A command pipe is commonly created when you want to execute an application from a command-line and pipe the output to another command-line. In the case of LabVIEW, we can use command pipes when we invoke a command-line application. We can pipe data into the application's input and pipe data out of its standard output.

On Linux, the Pipes VIs may be found on the Connectivity>>Libraries & Executables>>Pipes palette (see Figure 16.42). On Mac OS X, they are mysteriously missing from the palettes, but don't worryyou can find them all located in `vi.lib\Platform\system.llb`.

Figure 16.42. Pipes palette

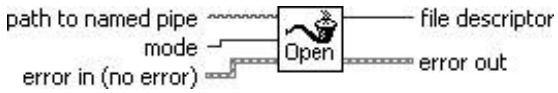


LabVIEW does not include support for pipes on Windows. You're probably thinking to yourself . . . "Whata LabVIEW feature that is supported on Linux and Mac OS X, but not on Windows?" Don't get too excited too quicklythe hard-working developers at OpenG have created a pipes library for Windows that is a drop-in replacement for the LabVIEW Pipes VIs. See Appendix C for more information on OpenG and how to obtain this library.

Now, let's take a look at the LabVIEW pipe VIs.

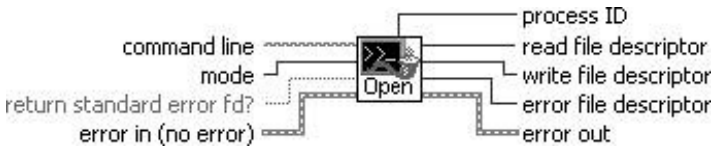
Open Pipe (Connectivity>>Libraries & Executables>>Pipes palette) opens a named pipe and returns file descriptor, which you pass to subsequent Pipes VIs (see Figure 16.43). A named pipe is a special file that can be used to communicate between separate system processes. Unlike a normal file, you must open a pipe in read mode, usually from another process or application, before you open the pipe in write mode so data written to the pipe can be passed to the reading process. Otherwise, an I/O error occurs and the open process fails.

Figure 16.43. Open Pipe



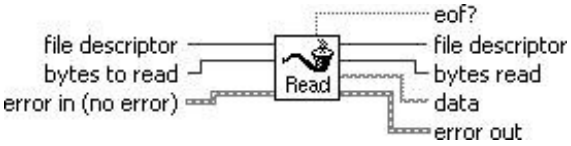
Open System Command Pipe (Connectivity>>Libraries & Executables>>Pipes palette) opens a pipe to a system shell command and returns file descriptors that you can pass to subsequent Pipes VIs (see Figure 16.44). This allows you to interactively pass data to a command-line application's standard input and read data back from its standard output.

Figure 16.44. Open System Command Pipe



Read From Pipe (Connectivity>>Libraries & Executables>>Pipes palette) reads a number of bytes from a pipe, returning the results in the data string output (see Figure 16.45). For this VI, you must have opened the pipe as a read pipe. The VI does not wait for data, so if the amount of data is not available, the VI returns any available data.

Figure 16.45. Read From Pipe



Write To Pipe (Connectivity>>Libraries & Executables>>Pipes palette) writes a data string to a pipe (see Figure 16.46). For this VI, you must have opened the pipe as a write pipe.

Figure 16.46. Write To Pipe

Close Pipe (Connectivity>>Libraries & Executables>>Pipes palette) closes a pipe (see Figure 16.47). Other processes reading the pipe will receive EOF (end of file).

Figure 16.47. Close Pipe

PREV NEXT

Talking to Other Computers: Network VIs

If you want to use LabVIEW to communicate over the network using low-level protocols, you can generally do so, but be aware that low-level networking takes a good deal of design and often more coding. You can use TCP/IP functions, for example, to communicate with any other computer, device, or program that also supports TCP/IP.

LabVIEW supports both the TCP and UDP, low-level Internet communication protocols, both of which are accessible from the Data Communication>>Protocols palette.

TCP/IP

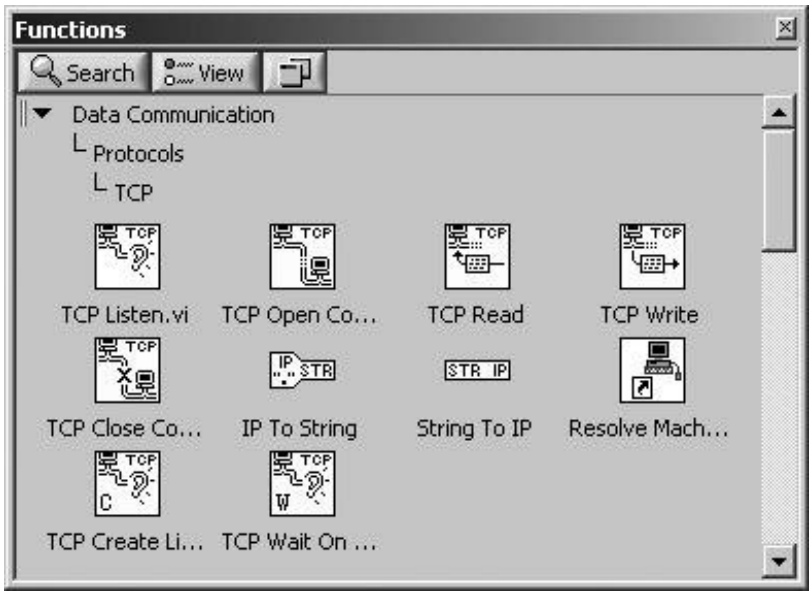
TCP/IP is the underlying protocol for the Internet and most internal networks.

Incidentally, TCP stands for *Transmission Control Protocol* and IP stands for *Internet Protocol*. IP divides your data into manageable packets called datagrams and figures out how to get them from A to B. Problem is, IP isn't polite and won't do any handshaking with the remote computer, which can cause problems. And, like bulk mail with the U.S. postal service, IP won't guarantee delivery of the datagrams. So they then came up with TCP which, added on to IP, provides handshaking and guarantees delivery of the datagrams in the right order (more like Federal Express, to follow the analogy).

TCP is a connection-based protocol, which means you must establish a connection using a strict protocol before transferring data. When you connect to a site, you have to specify its IP address and a port at that address. The IP address is a 32-bit number that is often represented as a string of four numbers separated by dots, like 128.39.0.119. The port is a number between 0 and 65535. You can open more than one connection simultaneously. If you're familiar with Unix or have used Internet applications, then this should all be old hat to you.

LabVIEW has a set of VIs, found under the Data Communication>>Protocols>> TCP palette, that let you perform TCP-related commands, such as opening a connection at a specified IP address, listening for a TCP connection, reading and writing data, etc. They are all fairly easy to use if your network is configured properly (see Figure 16.48).

Figure 16.48. TCP palette



Good examples for getting started with building your networked VIs are Simple Data Client.vi and Simple Data Server.vi, found in the Full version of LabVIEW. Both of their diagrams are shown in Figures 16.49 and 16.50.

Figure 16.49. Simple Data Client.vi block diagram

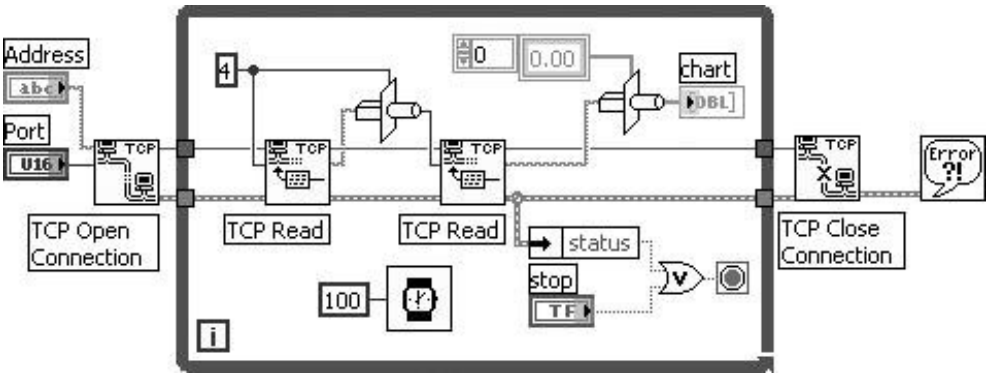
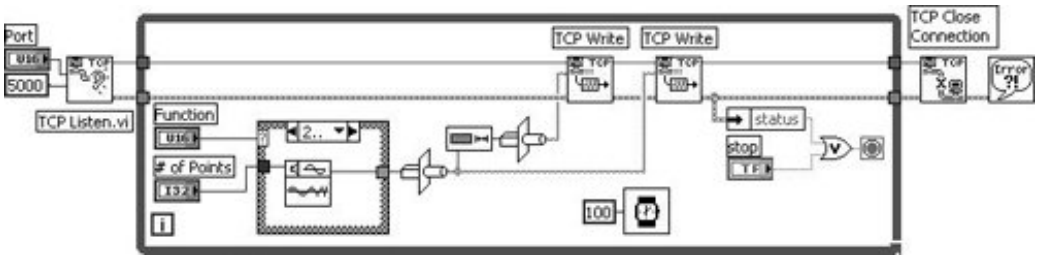


Figure 16.50. Simple Data Server.vi block diagram

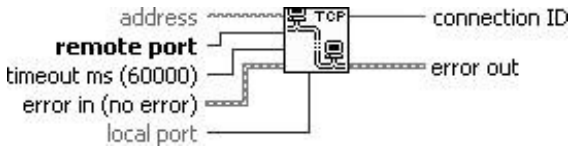
[View full size image]



You can learn quite a bit about writing client/server VIs by examining these diagrams. The basic process for the client is as follows:

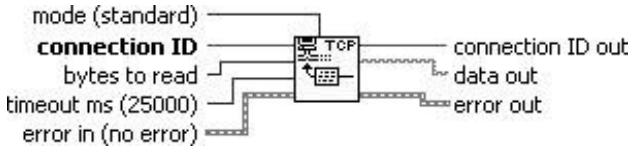
1. Request a TCP connection (see Figure 16.51). You can set a timeout to avoid hanging your VI if the server doesn't respond.

Figure 16.51. TCP Open Connection



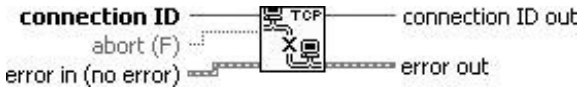
2. Read (or write, in other cases) data. Data is always passed as a string type (see Figure 16.52).

Figure 16.52. TCP Read



3. Close the TCP connection (see Figure 16.53).

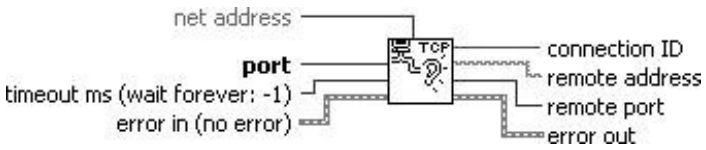
Figure 16.53. TCP Close Connection



The basic process for a server is as follows:

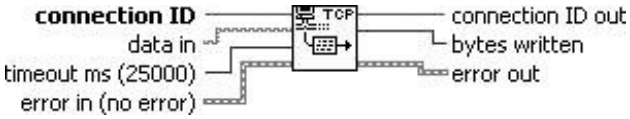
1. Wait for a connection (see Figure 16.54).

Figure 16.54. TCP Listen.vi



2. Write (or read, in other cases) data. Data is always passed as a string type (see Figure 16.55).

Figure 16.55. TCP Write



3. Close the connection.

Because all data over a TCP/IP network has to be passed as a string, you will need to convert your data to the LabVIEW string type. The easiest way to do this, as in the previous examples, is to use the Type Cast function (or the Flatten To String function). Just ensure that both the server and client know exactly what kind of data they're passing. If the server, for example, typecasts extended-precision floats to a string and the client tries to typecast the string into a double-precision number, the result will be garbage!

If you are looking for a generic way to pass data, consider passing the data as a variant that has been flattened to a string. Then you will always know what type the flattened data is (a variant). Then, after you unflatten the variant, you can use variant inspection tools to determine its typejust as we discussed in Chapter 14.

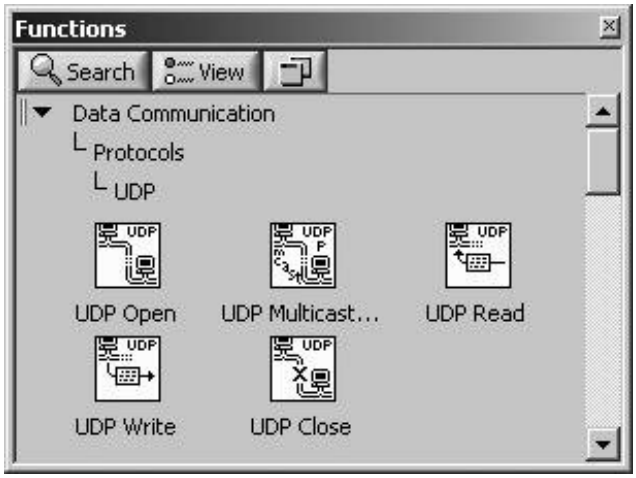
Networked applications are ideal when you need to write a program to control a large distributed system, such as in process control applications

UDP

UDP, which stands for *User Datagram Protocol*, is a lot like TCP, relying on IP addresses, except it is a "connection-less" protocol. What this means is that a server can broadcast data to a large number of clients without actually having to manage or be aware of connections between each one. That is why for UDP, there is no concept of a "listener" like there is in TCP.

LabVIEW provides support for UDP through the Data Communication>> Protocols>>UDP palette, shown in Figure 16.56.

Figure 16.56. UDP palette



UDP is a good choice for less network bandwidth and no handshaking when you need to broadcast non-critical data from LabVIEW.

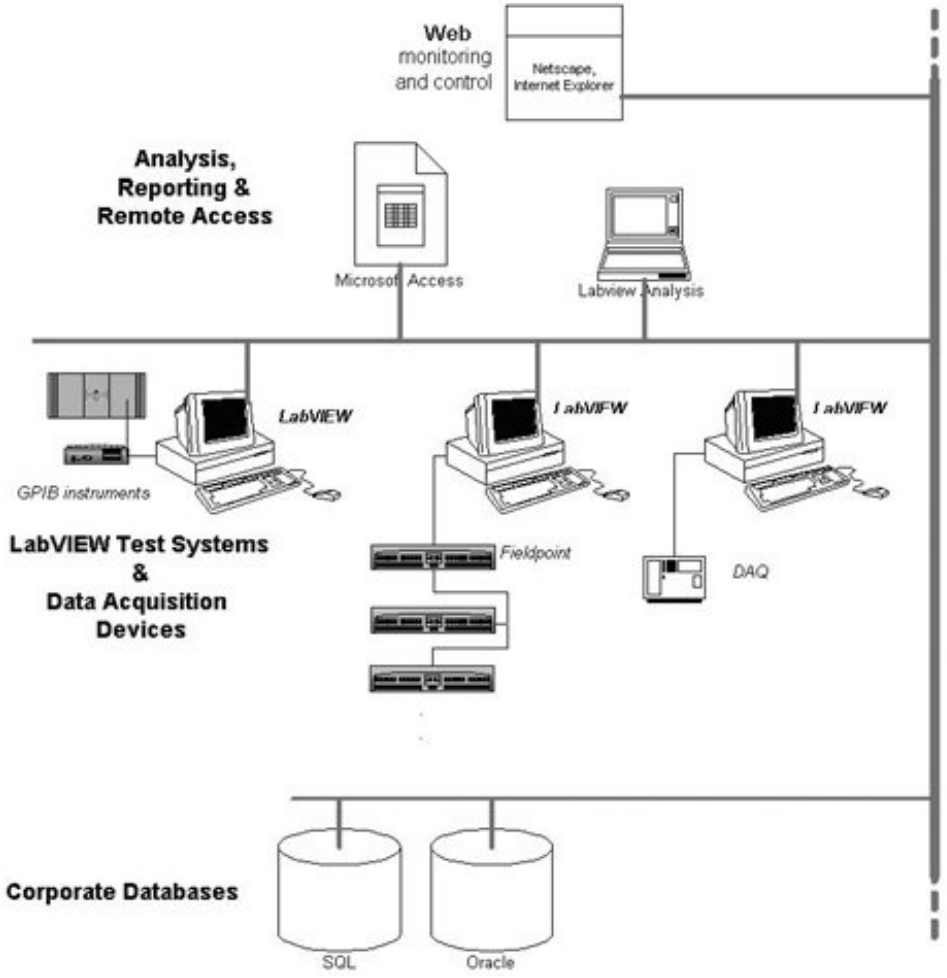
To see an example of UDP, run UDP Sender.vi and UDP Receiver.vi from the LabVIEW examples

Databases

For applications that require storing and processing lots of data, or where you need to be able to quickly sort and retrieve the data, you may need to design systems from the ground up with experts who understand the different tiers of an enterprise-level network. This often includes working with SQL databases such as Oracle or Sybase, tying together the LabVIEW labs to a common network, and implementing remote client access. Figure 16.57 illustrates this architecture.

Figure 16.57. Databases are an important part of large, distributed corporate applications.

[View full size image]



For example, suppose you had several test labs using LabVIEW to do functional testing on widget products. Adding Internet capability would allow you to monitor the current test status of any of the labs using a web browser. By tying a database into the network, you could not only obtain historical test information from the same web browser, but you could examine all kinds of interesting relationships. For example, you could compare average pass/fail times between different labs, or run a query comparing historical results with current real-time tests.

So, of particular importance in the enterprise system is the back-end database. In many cases, you will want to store your data acquisition test results in a database, rather than on your local drive. Databases allow far more efficient storage and retrieval of large amounts of data, particularly for situations of concurrent publishers and subscribers. In addition, databases allow programmers to create all types of queries that can examine relationships between elements in the database.

Although there are a variety of database vendors and brands, from the simple Microsoft Access to high-end Oracle systems, most of them can work with a common language known as SQL. SQL stands for Structured Query Language and is principally used to write queries that are sent to the database in order to retrieve the desired data. It's possible to use SQL statements with a variety of different databases, partly thanks to ODBC (Open DataBase Connectivity), a "glue" layer that abstracts SQL statements from an application and lets them work with a variety of database drivers.

Another way databases are used is to allow web access to the data remotely. By writing web server scripts (using CGI or ASP, for example), a remote user with a web browser can perform queries and pull up data or charts that are displayed right on the browser. If your LabVIEW-based application handles a large amount of data and you'd like to have web access to not only the real-time processes, but also historical data, consider tying a database to the LabVIEW system (as a publisher of data) and a web server script (as the subscriber of data). Your web scripts could even be written in LabVIEW as CGI VIs!

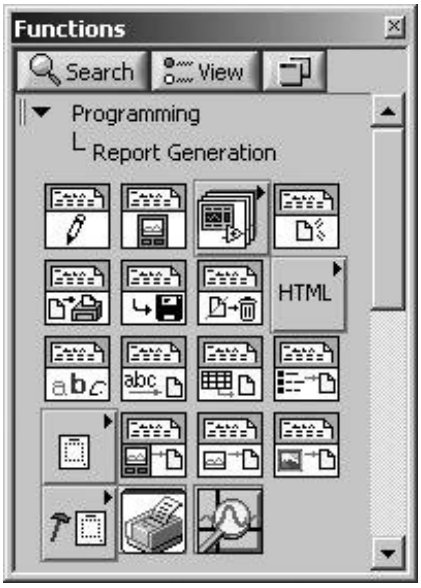
National Instruments sells a Database Connectivity Toolset for LabVIEW that has support for any ADO (ActiveX Data Objects) compliant database. (ADO supports both OLE-DB and ODBC, on the Windows platform.) This allows LabVIEW to read from and write to almost any SQL database. You can also use the free, open source LabSQL, which also supports ADO, for connecting to almost any database. A copy of LabSQL is on the enclosed CD, or you can download it from the LabVIEW Open Source Tools (LOST) site, http://jeffreytravis.com/lost.

PREV NEXT

Report Generation

Whether it is for your boss, your customer, or for taping to your refrigerator to show your family, there are a lot of good reasons to create a report. LabVIEW provides several VIs on the Programming>>Report Generation palette (shown in Figure 16.58) that allow you to generate reports and either save them or send them to a printer to make a hard copy.

Figure 16.58. Report Generation palette



The report generation VIs allow you to create reports in one of two formats:

- HTML file (that can be viewed in a web browser)
- RTF file (rich text format, which can be viewed and printed by any word processing application and most text editors)

You will need to specify which report format you want, when you create it.

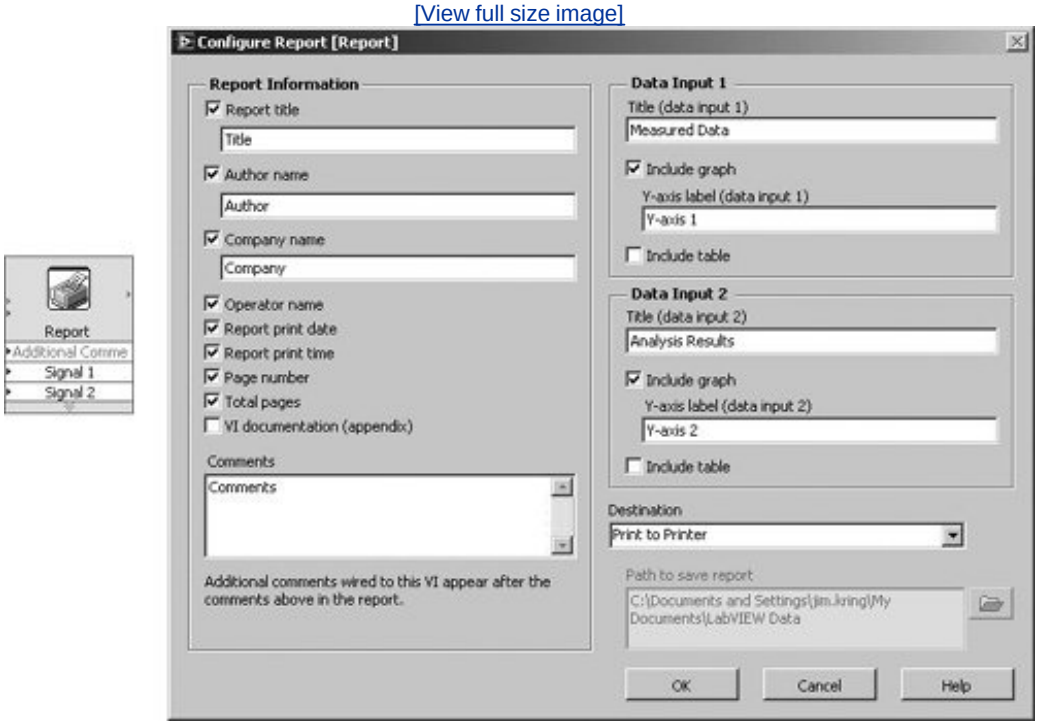
National Instruments also sells a LabVIEW Report Generation Toolkit for Microsoft Office. This provides VIs for programmatically creating and editing Microsoft Word and Excel reports from LabVIEW. This can come in very handy if your pointy-haired boss demands that all TPS reports be in a Microsoft Word format.

Express Report

A fast and easy way to generate a report from your measurement data is to use the Report Express VI, along with its configuration dialog (see Figure 16.59). This allows you to enter the report header information, configure whether to add graphs or tables of the measurement data, and specify the type of report (Destination).

Figure 16.59. The Report express VI along side its configuration dialog

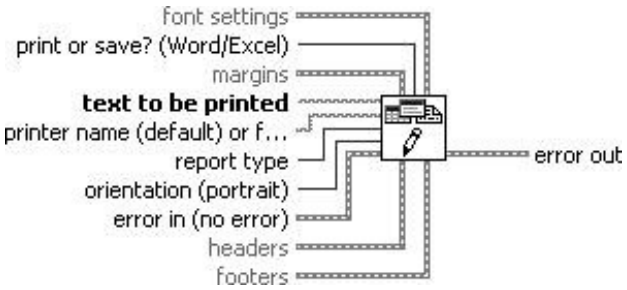
[View full size image]



Easy Text Report

Another quick way to create a report is by using Easy Text Report.

Figure 16.60. Easy Text Report



Easy Text Report (Programming>>Report Generation palette) allows you to pass in a block of text along with optional formatting information and print the report to a designated printer. Use tokens with this VI to generate information in the report, such as in the headers and footers. For example, you can insert a time stamp in the footers of the report. You cannot use tokens with Word and Excel reports.

The following list describes the tokens available when using Easy Text Report.

Token	*Description*
`<tab>`	For standard reports, LabVIEW indents the text to the next tab location. For HTML reports, LabVIEW interprets this token as ` `.
`<page>`	Current page number. LabVIEW ignores this token for HTML reports.
`<pages>`	Total number of pages. LabVIEW ignores this token for HTML reports.
`<pagenofm>`	Current page number with the total number of pages in the report. LabVIEW ignores this token for HTML reports. Example: `7 of 30`
`<shortdate>`	Current date in the form xx/xx/xx. Example: `10/5/98`
`<longdate>`	Current date in the form Day, Month and Date, Year. The month, date, and year order defaults to the date settings of your operating system. Example: `Monday, October 05, 1998`
`<time>`	Current time in the form Hour:Minute:Second AM/PM. Defaults to the clock settings of your computer. Example: `1:58:22 PM`

Token	*Description*
`<newline>`	Inserts a line break in the report.

Advanced Report Generation

For more advanced report generation, you will start out by calling New Report, which outputs a report reference that can be used with the other advanced report generation VIs. You will need to specify the report type (Standard or HTML) when you call New Report.

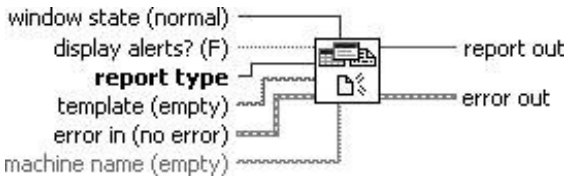
The sequence of operations for using the advanced report generation VIs is to:

1. Call New Report to create a new report, specifying the report type.
2. Call Set Report Font and the Report Layout palette VIs to configure the report styles.
3. Call the various Append . . . to Report VIs to add content to the report.
4. Call Print Report or Save Report to File.
5. Call Dispose Report to close the report.

Creating and Closing Reports

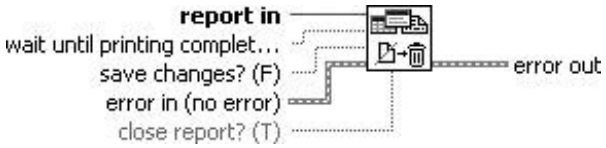
New Report (Programming>>Report Generation palette) creates a new report (see Figure 16.61). You must use this VI to create a report if you do not use the Easy Text Report VI.

Figure 16.61. New Report



Dispose Report (Programming>>Report Generation palette) closes the report and releases its interface, which saves memory. After the VI runs, you cannot perform further operations on the report. The Dispose Report VI should be the last VI you use when you create a report (see Figure 16.62).

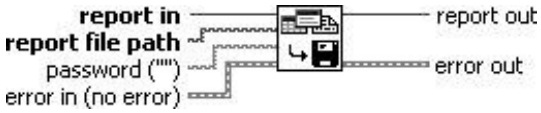
Figure 16.62. Dispose Report



Saving Reports

Save Report to File (Programming>>Report Generation palette) saves an HTML report to the file specified in report file path. You cannot use this VI with standard reports (see Figure 16.63).

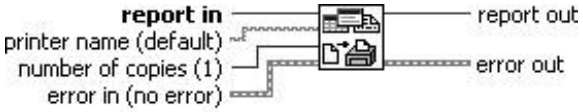
Figure 16.63. Save Report to File



Printing Reports

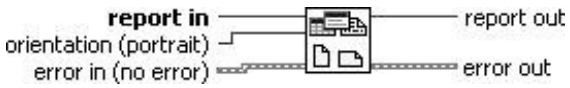
Print Report (Programming>>Report Generation palette) prints a report to a designated printer or to the default printer on the computer (see Figure 16.64).

Figure 16.64. Print Report



Set Report Orientation (Programming>>Report Generation>>Report Layout palette) determines whether the report is printed in landscape or portrait orientation (see Figure 16.65). You cannot use this VI with HTML reports. Use this VI prior to printing the report to ensure the correct orientation.

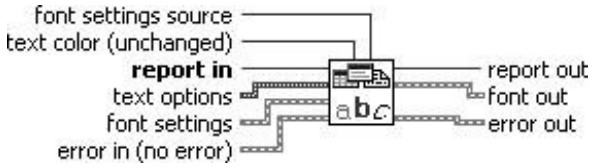
Figure 16.65. Set Report Orientation



Report Styles

Set Report Font (Programming>>Report Generation palette) sets the font properties of the report, including those in the headers and footers (see Figure 16.66). The available properties include italic, bold, strikethrough, underline, color, font name, font size, character set, and weight. The font you specify in the VI becomes the default font for the report.

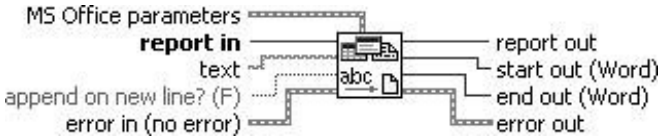
Figure 16.66. Set Report Font



Adding Report Content

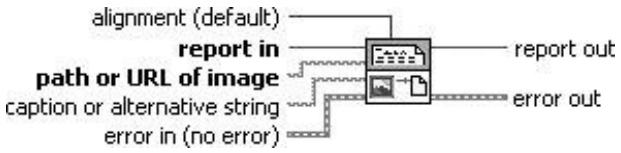
Append Report Text (Programming>>Report Generation palette) appends text to the selected report. The input into text must be a string. The selected report is the one passed into report in. You can append the text to the current position of the cursor in the report or on a new line (see Figure 16.67).

Figure 16.67. Append Report Text



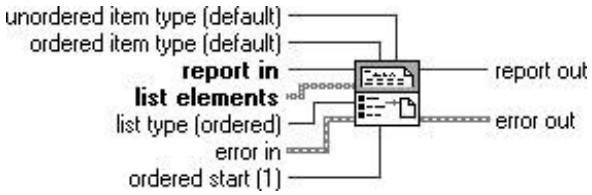
Append Image to Report (Programming>>Report Generation palette) For an HTML report, the VI embeds a link to an image into the report. For a standard report, the VI appends the image from a file (see Figure 16.68). Standard reports support `.bmp`, `.gif`, `.wmf`, `.emf`, and `.jpg` image files, but not `.png` files.

Figure 16.68. Append Image to Report



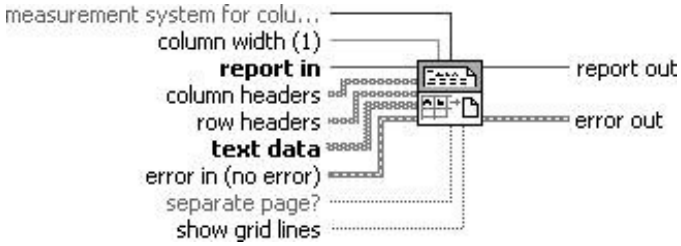
Append List to Report (Programming>>Report Generation palette) Adds a list of elements to the report. If you select ordered, the list is numbered or lettered. If you select unordered, the list is formatted with an item indicator or dash (see Figure 16.69).

Figure 16.69. Append List to Report



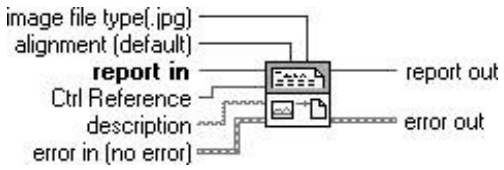
Append Table to Report (Programming>>Report Generation palette) Appends a 2D array to a report as a table with the given column width. The data type you wire to the text data input determines the polymorphic instance to use (see Figure 16.70).

Figure 16.70. Append Table to Report



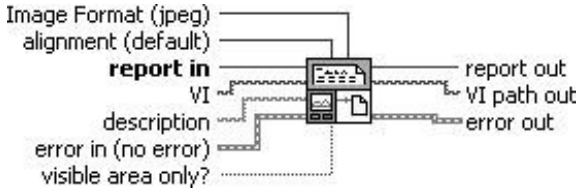
Append Control Image to Report (Programming>>Report Generation palette) Creates an image of the front panel object specified in Ctrl Reference and appends it to a report (see Figure 16.71). For an HTML report, the VI saves the image file to the temporary directory and adds a link to the image file in the report. For a standard report, the VI appends the image to the report.

Figure 16.71. Append Control Image to Report



Append Front Panel Image to Report (Programming>>Report Generation palette) Creates an image of the front panel of the VI you specify in VI and appends it to a report (see Figure 16.72). The data type you wire to the VI input determines the polymorphic instance to use. The VI also includes an instance whose connector pane is compatible with versions of the VI in LabVIEW 6.1 and earlier.

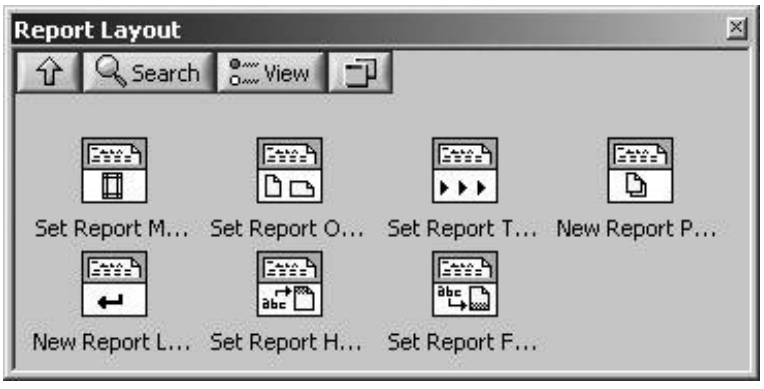
Figure 16.72. Append Front Panel Image to Report



Report Layout Palette

The Report Layout palette (see Figure 16.73) contains various VIs for configuring the layout of your report.

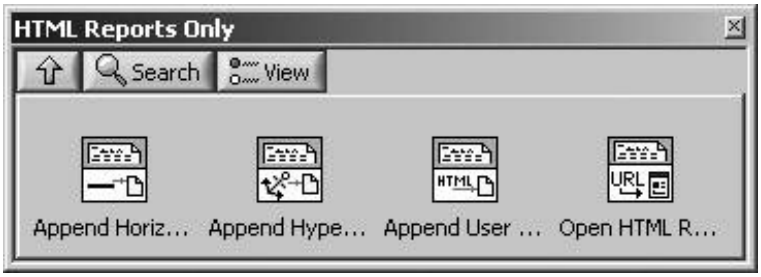
Figure 16.73. Report Layout palette



HTML Reports Palette

The HTML Reports palette (see Figure 16.74) contains various VIs specific to generating HTML reports.

Figure 16.74. HTML Reports Only palette

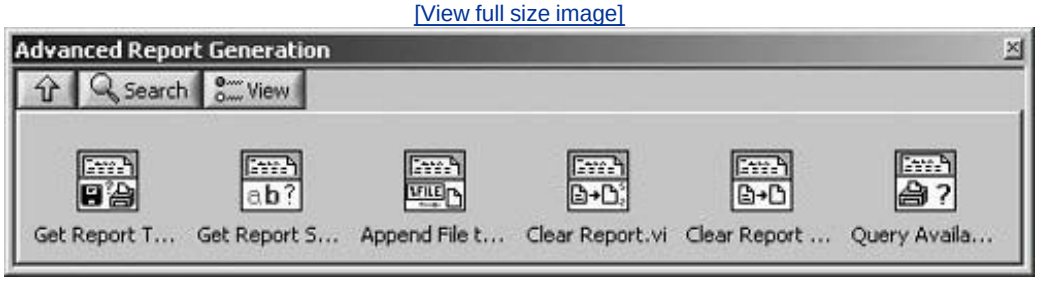


Advanced Report Generation Palette

The Advanced Report Generation palette (see Figure 16.75) contains various advanced report generation functions.

Figure 16.75. Advanced Report Generation palette

[View full size image]



PREV NEXT

Wrap It Up!

In this chapter, we looked at the role of the Internet and networks, and the different connectivity features LabVIEW offers: web-enabled VIs, network variables, XML, TCP/IP, UDP, ActiveX, AppleEvents, pipes, databases, and report generation.

We saw how LabVIEW's built-in web server allows you to easily monitor or publish your VIs on the Web. With the LabVIEW web server, you can simply type a URL in the web browser and instantly see an image or control a VI.

The new shared variables (on Windows) and the older DataSockets are protocols for effective network variables that easily share data between NI-based applications. With network variables, you can easily hook any front panel control or indicator into the shared variable, so other VIs or applications can publish or subscribe to that control. You can also use the DataSocket block diagram functions to share data.

Lower-level protocols, like TCP/IP and UDP, as well as .NET and ActiveX on Windows, are supported in LabVIEW. If you know about these protocols, you can use them for writing network or inter-application communication VIs.

We saw the importance of databases in the enterprise. Often you'll want to use a database to write and retrieve your LabVIEW data. You can use tools like LabSQL or the Database Connectivity Toolset to accomplish this.

And finally, we took a look at the report generation VIs for creating, saving, and printing reports in either HTML or RTF format.

17. The Art of LabVIEW Programming

Overview

This chapter focuses on techniques, tips, and suggestions for making better LabVIEW applications. We start with the appearance of the front panel of your applicationmaking it look as cool and user-friendly as possible. The whole concept of an intuitive, appealing graphical user interface (GUI) is embedded in LabVIEW. We'll show you a few tips and techniques for taking it a bit further with suggestions on panel arrangement, decorations, customized controls, dynamic help windows, and more. Some common programming problems and their solutions will be presented. We will also cover issues such as performance, memory management, and platform compatibility to help you write better VIs. Finally, we'll examine some general aspects of good programming techniques and style, which you will find especially useful in managing large LabVIEW projects.

Goals

- Become familiar with general guidelines and recommendations for creating an aesthetically pleasing, professional-looking graphical interface
- See how you can import external pictures onto your front panel and into picture rings
- Build custom controls using the Control Editor and learn what XControls are
- Learn to dynamically open and close the Help window
- Become familiar with some nifty solutions to common LabVIEW programming challenges (the kind that will make you go, "a-ha!")
- Be able to boost performance, gobble less memory, and make your VIs platform-independent when you need to
- Know some proposed guidelines for writing an awesome application
- Become the coolest LabVIEW programmer in town

Key Terms

- Decorations
- Custom controls
- Control Editor
- Type definition
- Strict type definition
- XControl
- Importing pictures
- Custom help
- Aesthetics
- Common questions
- Memory usage
- Performance
- Platform dependency
- Good style

Why Worry About the Graphical Interface Appearance?

In our culture, image counts for a loteven in software. People are often more impressed by what a program looks like on the screen than how it actually works. I can think of at least a few reasons why you would want to polish and improve your graphical user interface (GUI): 1) To impress and convince someone else (your supervisor, customers, spouse) about the quality of your software and its goals; 2) To make your software more intuitive, easier, and fun to use for the end user (yes, having fun is important!); 3) It's easier than doing actual work (why can't you have a little fun, too?). And therein lies the beauty of LabVIEWeven if you're a novice user, you can put together an impressive graphical interface on the front panel (never mind that it doesn't do anything yet) before a guru C programmer can blink.

LabVIEW's front panel objects are already pretty cool-looking: knobs, slides, LEDs, tree controls, tab controls, splitter bars, and so on. But this chapter will teach you a few more tricks for organizing and putting your objects on the front panel in such a way to create an even better interface. You will also see how to add online help so that the end user can figure out what each control does without a manual. Making a good GUI is not just about aestheticsit's also about saving the user time and effort.

Look at the VI shown in Figure 17.1, Temperature System Monitor. It doesn't look too bad. But almost anybody would agree that this same VI with the interface depicted in Figure 17.2 would be easier to work with and certainly have greater visual appeal.

Figure 17.1. A user interface that doesn't look too bad

[View full size image]

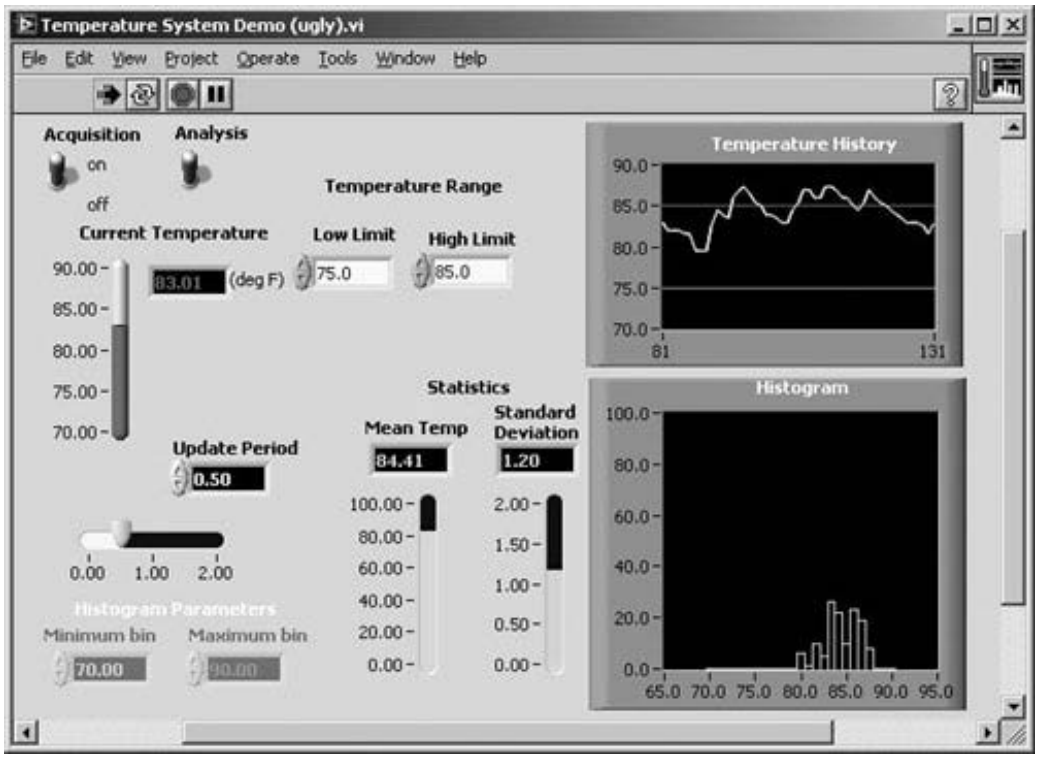
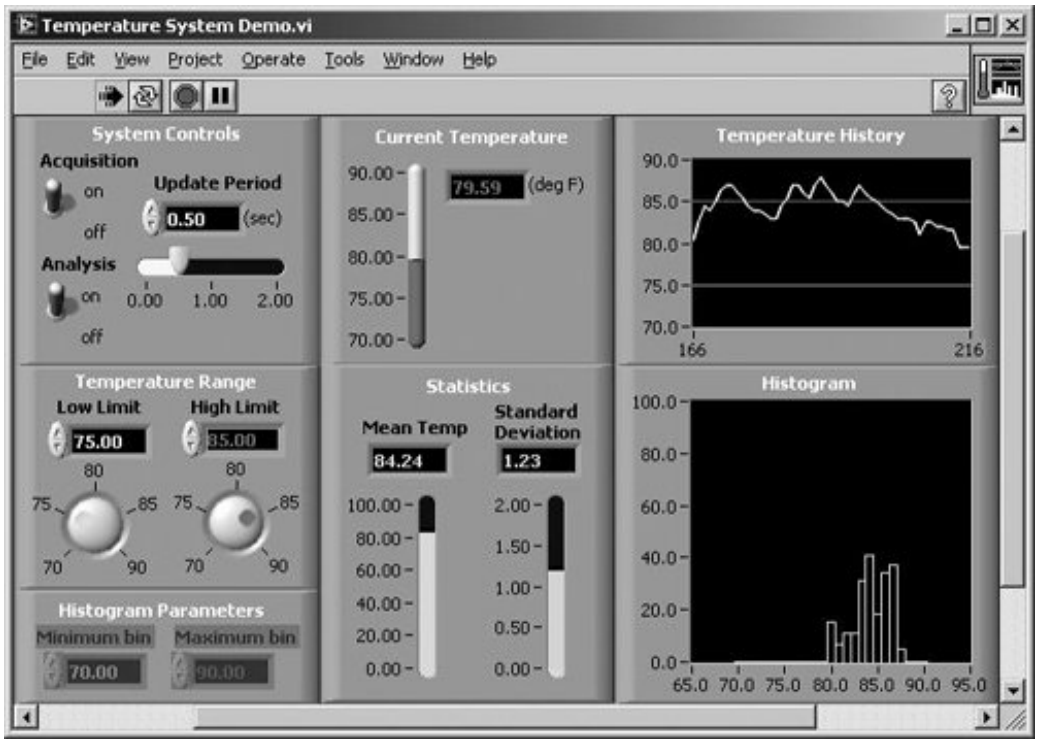


Figure 17.2. A user interface that looks better

[View full size image]


Temperature System Demo.vi
File Edit View Project Operate Tools Window Help
System Controls
Acquisition
on
off
Update Period
0.50
(sec)
Analysis
on
off
0.00
1.00
2.00
Current Temperature
90.00
85.00
80.00
75.00
70.00
79.59
(deg F)
Temperature History
90.0
85.0
80.0
75.0
70.0
166
216
Temperature Range
Low Limit
High Limit
75.00
85.00
80
75
85
70
90
Statistics
Mean Temp
84.24
Standard Deviation
1.23
100.00
80.00
60.00
40.00
20.00
0.00
2.00
1.50
1.00
0.50
0.00
Histogram
100.0
80.0
60.0
40.0
20.0
0.0
65.0 70.0 75.0 80.0 85.0 90.0 95.0
Histogram Parameters
Minimum bin
Maximum bin
70.00
90.00

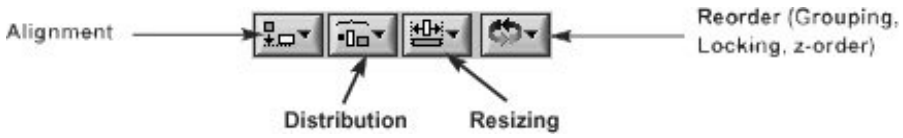
PREV NEXT

Arranging, Decorating, Resizing, Grouping, and Locking

One way to improve the appearance of your front panel is to organize the objects by deciding where to place them on the panel. You can do this by aligning and distributing your objects evenly and by grouping sets of objects that are logically related onto or inside a decoration.

LabVIEW provides you with the ability to *align*, *distribute*, *resize*, *group*, *and lock* objects, much like a drawing program, as we saw in Chapter 4, "LabVIEW Foundations." You have a variety of choices for how to align your objects. Distribution of objects refers to the spacing arrangement between objects. Resizing allows you to select multiple objects and adjust their height and/or width to that of the smallest or largest object in the selection of objects. Grouping objects allows you to easily move a set around without disturbing their relative arrangement. And locking objects helps keep them from being accidentally selected or resized. In addition, you can change the z-order, or depth placement (what's in front and what's behind), of any object. All of this graphical control is done with the toolbar set you'll find on the VI window (see Figure 17.3).

Figure 17.3. Align, Distribute, Resize, and Reorder toolbar buttons

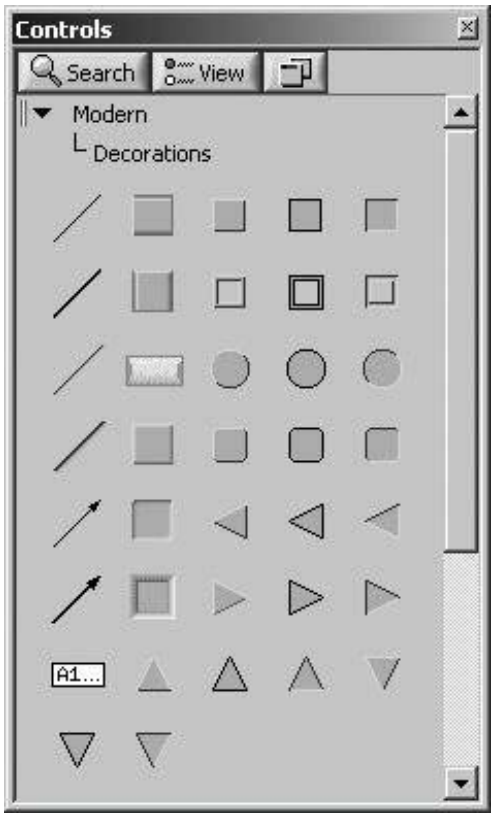


You will quickly realize that LabVIEW considers an object's label to be part of the object when it performs alignment and distribution (but not for resizing) operations. If this is not the behavior that you desire, you can temporarily hide the offending object labels by selecting Visible Items>>Label from each object's pop-up menu, and then selecting that option again for each object to restore the labels.

Also, remember that the label of an object can be placed anywhere you want to, but it remains attached to (and part of) that object.

By now you may have noticed the Modern>>Decorations subpalette in the Controls palette (see Figure 17.4).

Figure 17.4. Decorations palette



These front panel decorations are just thatthey don't do anything, and they have no corresponding block diagram terminal. You can use these boxes, lines, frames, and so on, to encapsulate and group your front panel controls and indicators into logical units. You might also use them to approximate the front panel of an instrument more closely.

Often you'll have to make use of the previously mentioned z-order functions, such as Move To Back, to place the decorations "behind" your controls. The reason for these commands is obvious once you create a decoration to enclose a set of controls: decorations aren't normally transparent, so if you place a decoration on the front panel *after* you create the controls, the decoration will obscure them because it is at the "front."

Clicking on a decoration is not the same as clicking on a blank part of the front panel. When using decorations behind controls, it's easy to unintentionally select the decoration with the cursors in the edit mode. The exception to this warning is the set of "frame" decorations, which are only "clickable" on the edge of the frame.

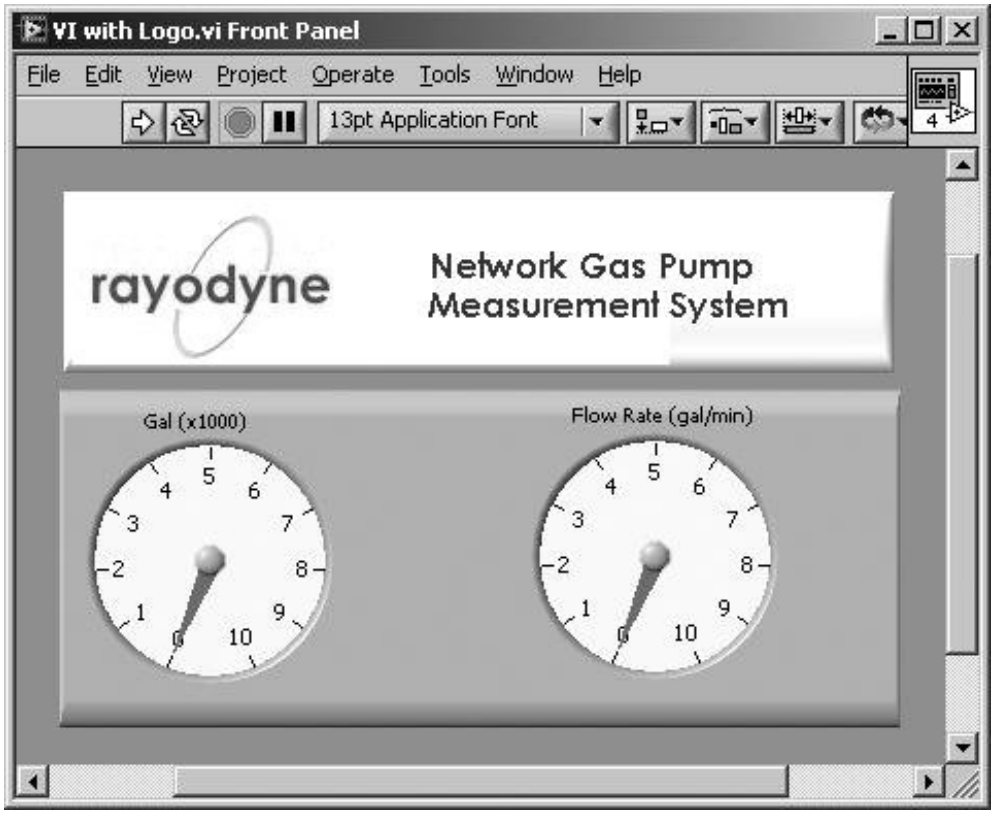
Once you are happy with the size and location of a frame (or other) front panel decoration, lock it in place so that it cannot be moved. Do this by selecting the decoration and then click the Reorder button on the toolbar and select Lock. To unlock, follow the same process, but instead select Unlock.

PREV NEXT

Vive l'art: Importing Pictures

You can paste pictures right into LabVIEW and include them on your front panel (or block diagram, but this is not a common practice). For example, you might want to make a block on your instrument that has your company logo (see Figure 17.5). Or you might want to be a little more elaborate, like adding piping and valve pictures to represent some process control loop (you can actually make your valve pictures Boolean controls, as explained in the next section).

Figure 17.5. Front panel with a company logo that was imported into LabVIEW



To import a picture into LabVIEW, use the Edit>>Import Picture from File . . . dialog to select the image file and load it into LabVIEW's clipboard.

In Windows and Mac OS X, you can also import images into LabVIEW by copying a picture directly from another application in which it's open. However, this will cause the image to pass through the operating system's clipboard and is not advisedthere is a potential for loss of information, reformatting, or distortion of the image. It is recommended that you always use the Edit>>Import Picture from File . . . mechanism for importing images into LabVIEW.

The following are lists of supported image formats, for each platform:

All Platforms support these formats:

- PNG, including support for transparency
- GIF, including support for transparency and animation
- JPG

On Windows, it supports these additional formats:

- BMP (Windows Bitmap File)
- CLP (Microsoft Windows Clipboard)
- EMF (Enhanced Meta File)
- WMF (Windows Meta File)

On Linux, it supports these additional formats:

- XWD (X Window System Dump)

In Windows and Mac OS X, you can bypass the Edit>>Import Picture from File . . . menu option by simply dragging and dropping a supported image type from File Explorer (Windows) or Finder (Mac OS X) onto a VI's front panel. The image will then appear on the front panel as a decoration.

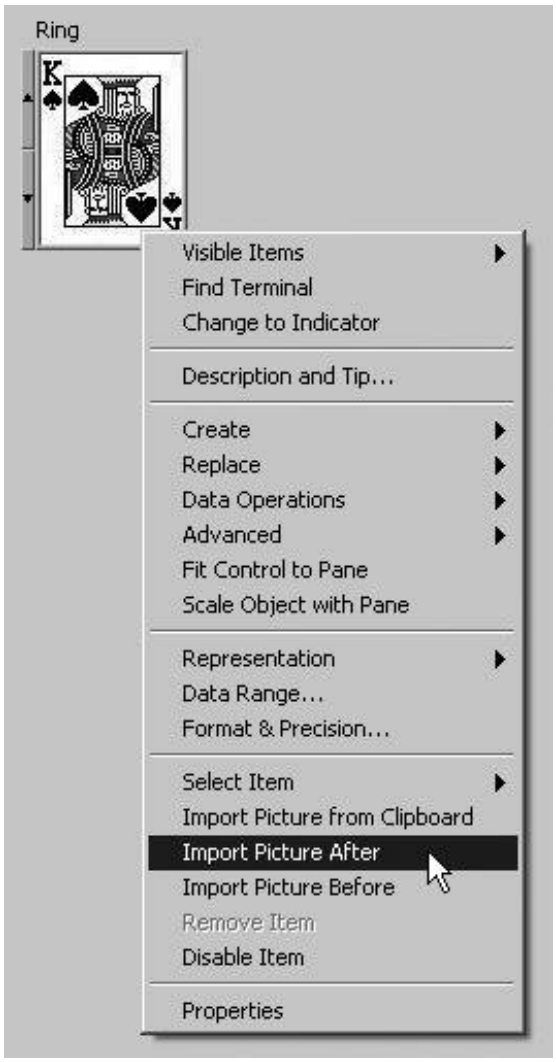
In addition to blank areas on the front panel (or block diagram), another place where you can put pictures is in a picture ring (Pict Ring) or a text and picture ring (Text & Pict Ring), available from the Modern>> Ring & Enum palette. Two examples are shown in the following illustration (see Figures 17.6 and 17.7). Using the picture rings allows you to get very creative in presenting the user with a customized *graphical* set of options that you can index and track.

Figure 17.6. Picture ring

Figure 17.7. Text and picture ring

To add a picture to a picture ring, first import your picture (using the Edit>>Import Picture from File . . . menu option) to the clipboard. Then choose Import Picture from the pop-up menu on the Picture Ring control. To add more pictures, choose either Import Picture After or Import Picture Before from the pop-up menu. Choosing Import Picture overwrites the current picture (see Figure 17.8).

Figure 17.8. Importing a picture into a picture ring



In Windows and Mac OS X, you can bypass the Edit>>Import Picture from File . . . menu option by simply dragging and dropping a supported image type from File Explorer (Windows) or Finder (Mac OS X) onto a Picture Ring control. The image will replace the current image. Pop up on the picture ring and select Add Item Before or Add Item After to create a new "blank" item.

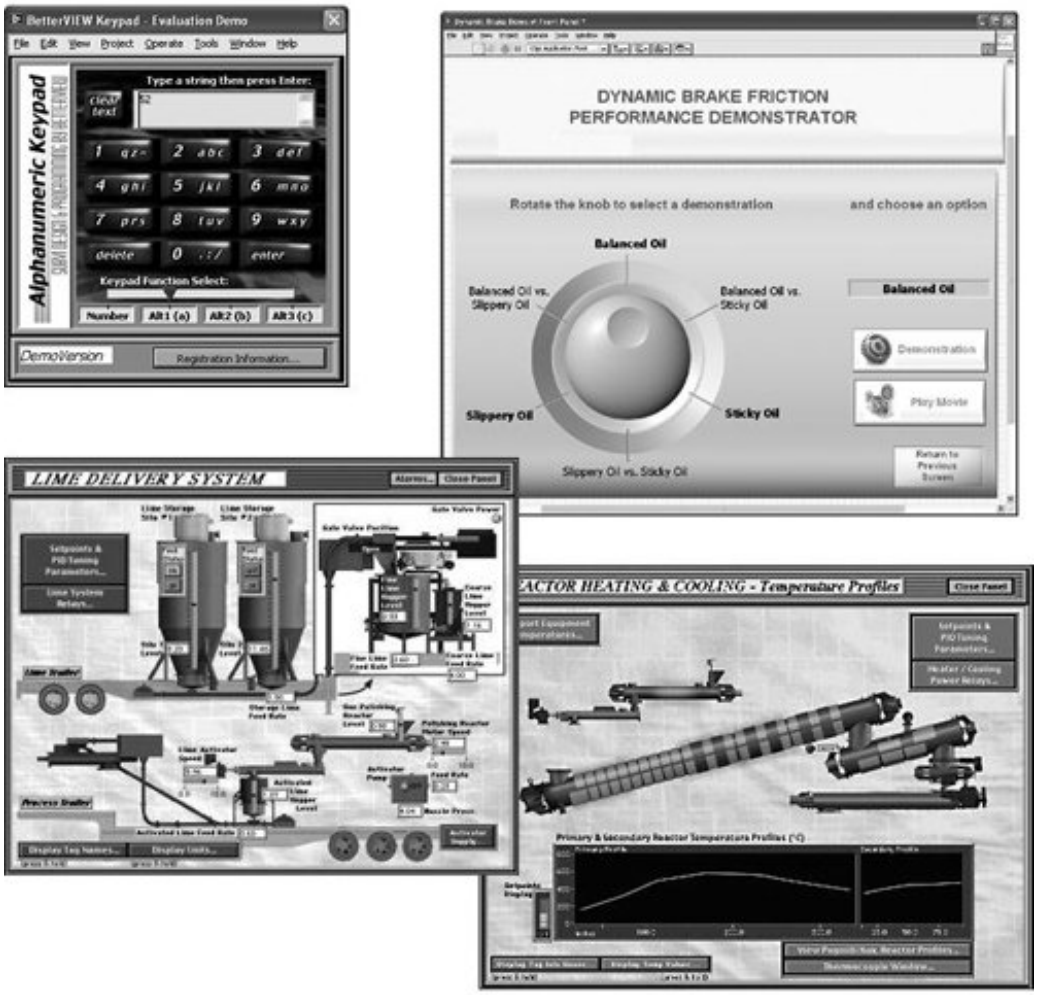
Custom Controls and Indicators

Q1: What do all the VI front panels shown in the following figure have in common?

A1: They all use custom LabVIEW controls! And look darn cool, too (see Figure 17.9).

Figure 17.9. Custom LabVIEW controls in front panels.Top-left and bottom two images courtesy of Dave Ritter, http://www.bettervi.com

[View full size image]



In Chapter 13, "Advanced LabVIEW Structures and Functions," we learned about type definitions, a kind of custom control. Type definitions usually focus on the underlying data type, but with custom controls, you can also customize the appearance of the control.

You can customize controls and indicators to make them better suited for your application while displaying a more impressive graphical interface. In the previous examples, you might want to display the temperature stages at various points in a process to represent the real-world scenario. Or you might want to turn on and off a valve by clicking on a valve picture control above.

You can save a custom control or indicator ("control" from here on, for simplicity) in a directory or LLB file, just as you do with VIs and globals. LabVIEW uses the ".ctl" file extension for custom control files. You can use the saved control in other VIs, as well as in the one in which it was created. It's possible to even create a master copy of the control if you need to use it in many places in the same VI by saving the control as a *type definition*. When you make changes to a type definition, LabVIEW automatically updates all the VIs that use it! That's a huge time saver and an absolute necessity for large LabVIEW projects.

You may also want to place a frequently-used custom control in the Controls palette. Do this by selecting Tools>>Advanced>>Edit Palette Set from the menu, or just put it in the `./user.lib/` directory, and it will appear in the User Controls palette the next time you restart LabVIEW.

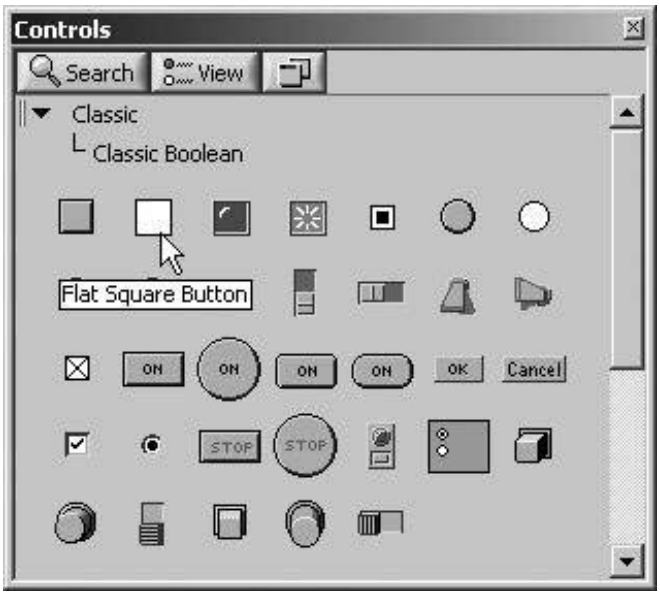
OK, so how do you create a custom control? We'll start with a very simple example.

Activity 17-1: Custom Controls

Usually, you will want to import pictures for your custom control, so first of all have your picture files available, maybe along with a graphics program. LabVIEW doesn't have any sort of picture editor. When you create a custom control, you always base it on the form of an existing control, such as a Boolean LED or a Numeric Slide control. In this activity, we'll create a custom Boolean that looks like a valve that is open or closed.

1. Place a Flat Square Button control (available on the Classic>>Classic Boolean palette) on a front panel, and select it (see Figure 17.10).

Figure 17.10. Classic Boolean palette



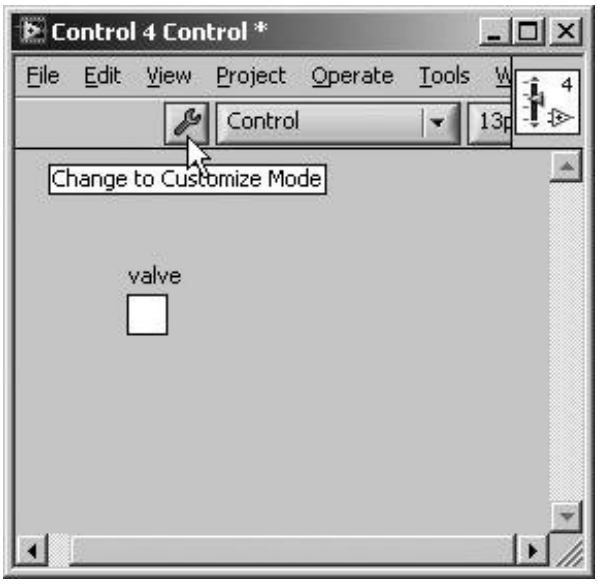
For this activity, we are using a Classic (2D) button because it only has one "part"a part is an individual item that you can edit in the Customize mode of the Control Editor dialog. Because our valve only has one part, this choice makes sense. Modern (3D) buttons have multiple "parts," each of which may be replaced with an image using the process detailed in this activity.

2. Launch the Control Editor by selecting Customize Control . . . from the Edit menu (or select Advanced>>Customize . . . from the control's pop-up menu).

3. The Control Editor window will show the Boolean.

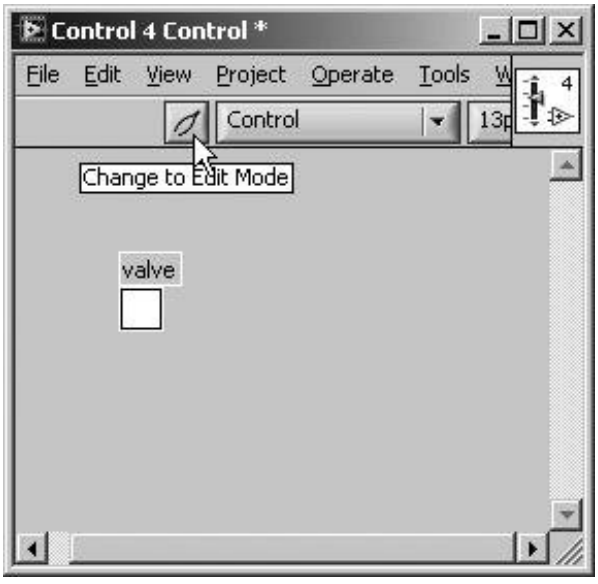
4. In Edit mode, the Control window works just like the front panel. You can pop up on the control to manipulate its settingsthings like scale, precision, etc. (see Figure 17.11).

Figure 17.11. The Control Editor being changed to Customize Mode



5. In Customize mode, which you access by clicking on the tool button, you can resize, color, and replace the various picture components of the control (see Figure 17.12).

Figure 17.12. The Control Editor being changed to Edit Mode

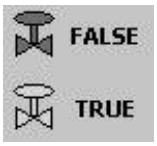


6. In Customize mode, pop up on the Boolean and select Import Picture from File . . . and select the picture file of a closed valve or use the one provided on the CD (named `closed.png`). as shown in Figure 17.13.

Figure 17.13. Your Boolean control after importing **`closed.png`**

7. Repeat step 6 for the TRUE case of the Boolean, using a different valve picture (see Figure 17.14). The TRUE (open) valve picture is also on the CD (`open.png`).

Figure 17.14. Your Boolean control shown in both the FALSE and TRUE states after importing **open.png**



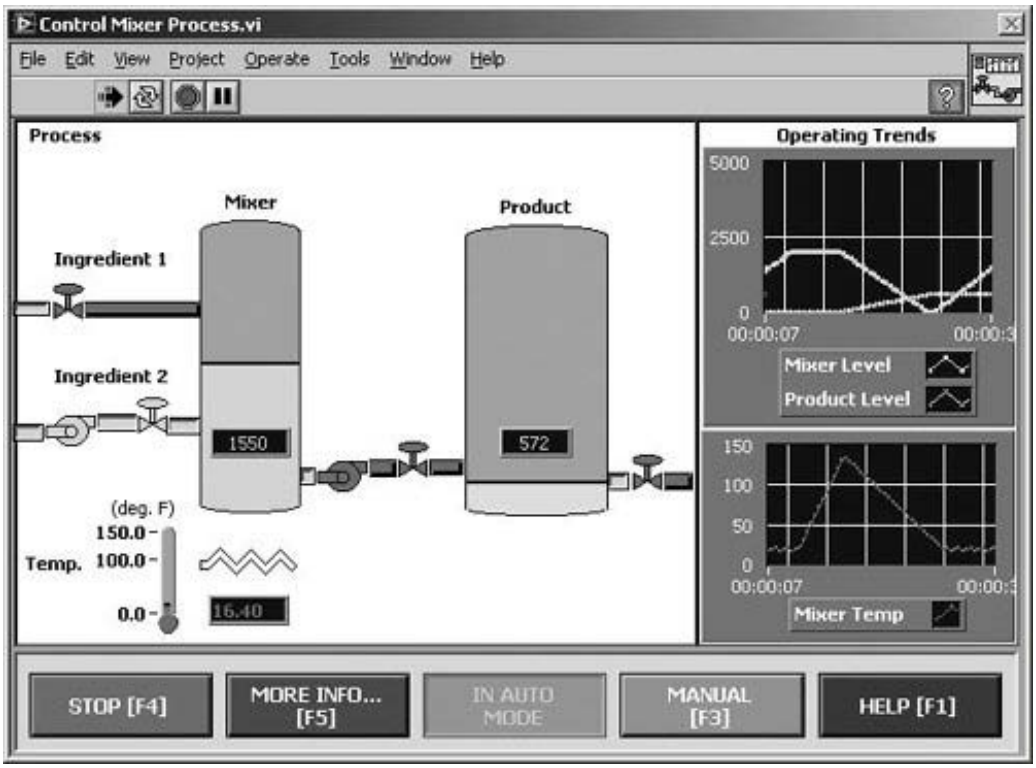
The valve image files included on the CD are of the PNG format and have been configured to have a transparent background. This allows you to use your valve button on the front panel of VIs that have any coloryou will not see the background of the valve image. This is a very cool LabVIEW feature!

8. Save the custom control by selecting Save from the File menu. LabVIEW uses the "`.ctl`" file extension for custom controls.

9. The front panel shown in Figure 17.15 is found in the examples of the full version of LabVIEW (`examples\apps\controlmix.llb\Control Mixer Process.vi`). This front panel uses the same Boolean controls you just created, as well as some others.

Figure 17.15. Control Mixer Process.vi front panel

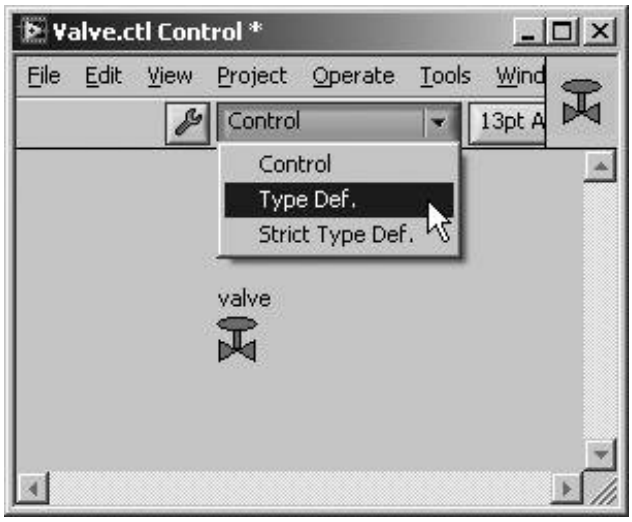
[View full size image]



An important concept to understand is that you cannot change the behavior of a control; you can only modify its appearance. This also means that generally you can't change the way a control displays its data (a custom control based on a slide will always have something that slides in one dimension, for example). Despite these limitations, you can produce some fancy graphical interfaces, especially if you're artistically inclined and willing to experiment.

If you want to create a master copy of your control so that all the VIs that contain it are automatically updated when you make a change to the master copy, select Type Def. from the ring on the Control Editor's toolbar (see Figure 17.16).

Figure 17.16. Configuring Valve.ctl to be a Type Definition



A type definition forces the control data type to be the same everywhere it is used, but different copies of the type definition can have their own name, color, size, and so on. This is useful because only the master copy can be modified in behavior, thus preventing accidental changes. A *strict type definition* forces almost everything about the control to be identical everywhere it is used.

Finally we'll mention one very powerful type of customized control: the XControl. XControls in LabVIEW let you not only create custom interfaces, but let you *embed custom behaviors* by writing LabVIEW code for the XControl. For example, you could create an XControl that is like a CD Player interface. Among other things, you could define that the "stop" button works whenever the "play" button is active and pressed, and that the "play" button resets the "stop" button.

XControls are beyond the scope of this book but for more information, check out the LabVIEW examples and documentation on this very nifty feature.

PREV NEXT

Adding Online Help

LabVIEW's Help window is extremely useful when you need learn something in a hurry; most people come to depend on it for wiring their diagrama very good programming habit to develop. You can make your own application just as easy to learn by adding your own entries for the Help window, as well as links to a hypertext Help document. Three levels of customized help are available:

1. Tooltip Help: Also called "hesitation help," it's the text in a yellow box that shows up when you move the mouse cursor over a control and hold it there.

2. Window Help: The comments that appear in the Help window that describe controls and indicators as you move the cursor over them.

3. Online help in a Help (hypertext) document: You can programmatically create a link to bring up an external help file.

Providing customized help in the Tooltip format and Help window is fairly easy. We covered this briefly in Chapter 5, "Yet More Foundations." To do this, choose Description and Tip . . . from the pop-up menu of a control or indicator. Enter the tip and description in the dialog box, and click OK to save it. LabVIEW displays this description whenever you have the Help window open and move the cursor over the front panel object (see Figures 17.17 and 17.18).

Figure 17.17. The Description and Tip dialog being used to edit the description and tool-tip for the **alarm** numeric control

[View full size image]

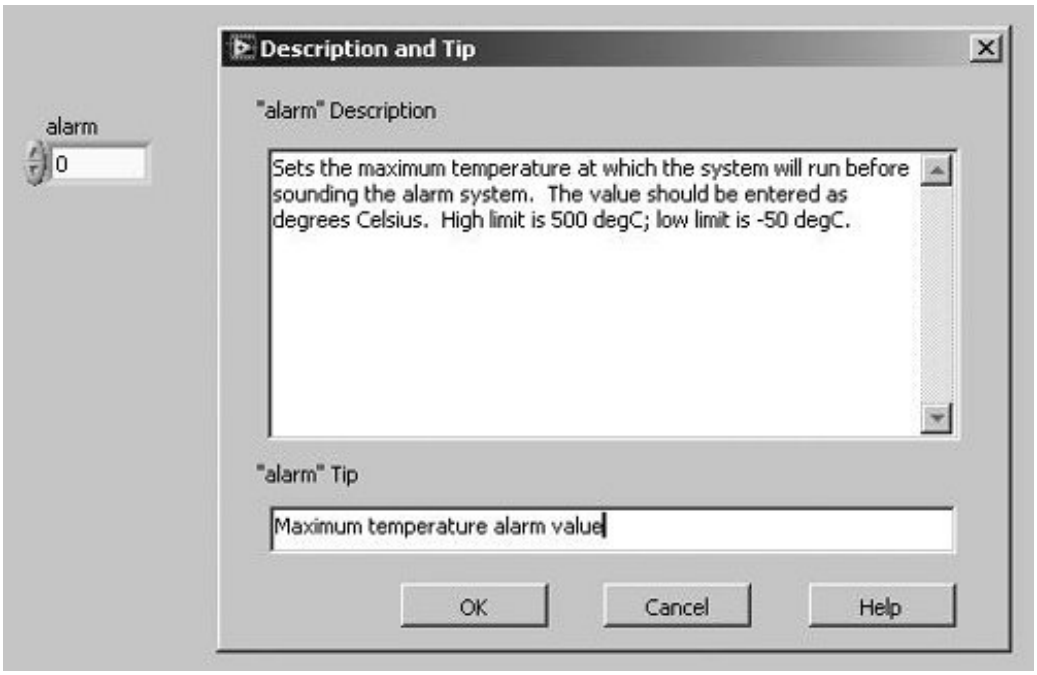
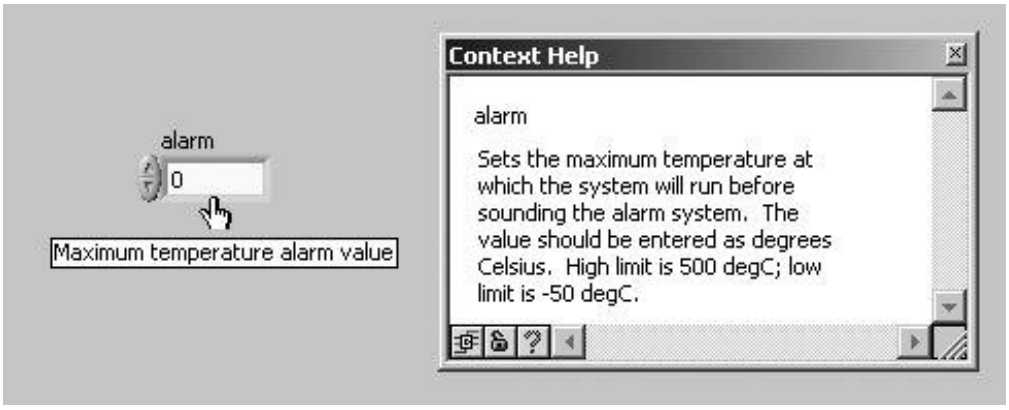


Figure 17.18. The Context Help window showing the description of the **alarm** control as the mouse cursor is hovered over it



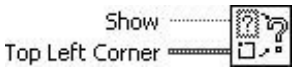
If you document all your front panel controls and indicators by including their descriptions, the end user can open the Help window and quickly peruse the meaning of your front panel objects.

To provide a description for the whole VI, enter information in the text box that appears when you choose Documentation from the VI Properties... menu. These comments will show up in the Help window, along with the wiring diagram, whenever the cursor is positioned over the VI's icon in the block diagram of another VI or in the upper-right corner of the VI's own front panel or block diagram. It will also show up in the Help window when the user hovers over the VI in the Functions palette.

You can also bring up, position, and close the Help window programmatically. Use the Control Help Window and the Get Help Window Status functions, available from the Help subpalette of the Programming>>Dialog & User Interface palette.

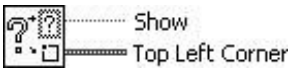
Control Help Window (Programming>>Dialog & User Interface>>Help palette) controls the position and visibility of the Help window. The Boolean Show input closes or opens the Help window; the cluster input consists of two numerical indicators that determine the top and left pixel position of the window (see Figure 17.19).

Figure 17.19. Control Help Window



Get Help Window Status (Programming>>Dialog & User Interface>>Help palette) returns the state and position of the Help window (see Figure 17.20).

Figure 17.20. Get Help Window Status



The more advanced help method, calling a link to an external, or *compiled*, help file, (which is usually written in HTML or RTF) is more elaborate. However, these are the same professional help files you see in commercial applications, and are very useful for an application you plan to distribute.

You can use the HTML or RTF files LabVIEW generates to create your own compiled help files. Complete the following steps to create a compiled help file:

1. Generate HTML or RTF files from the VI documentation. You can do this by choosing Print... from the File menu in the top-level VI and, in the ensuing wizard dialog, choosing the "VI Documentation" option. After you choose this, you will get the option of printing to a printer, HTML, RTF, or plain text destination.

2. You will need to choose either HTML or RTF as the destination.

    - Generate HTML files if you want to create a Windows-compiled HTML Help file or Mac OS Apple Help.
    - Generate RTF files if you want to create a Windows WinHelp or Linux HyperHelp file.

3. Compile the source HTML or RTF documents into one file. You can use one of the following third-party utilities to compile the documents, but several other utilities are also available:

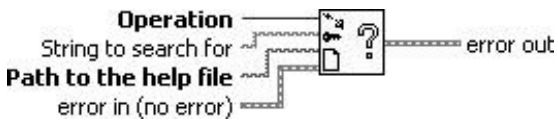
    - Windows You can use Microsoft HTML Help Workshop to compile HTML Help files and use Microsoft Help Workshop to compile WinHelp files. (You can also use Doc-to-Help from ComponentOne to easily convert Microsoft Word documents into compiled or HTML help.)
    - Mac OS X You can use Apple Help to view HTML files as a help system.
    - Linux You can use the HyperHelp compiler from Bristol Technology Incorporated to compile QuickHelp files.

4. Link from the VIs you documented to the compiled help file. You can do this by popping up on the VI's icon, selecting VI Properties>>Documentation, and using the "Browse..." button to link to the compiled help file. For a fuller description of this functionality, see the LabVIEW documentation.

To call these help files programmatically, you can use the remaining function in the Programming>>Dialog & User Interface>>Help subpalette.

Control Online Help (Programming>>Dialog & User Interface>>Help palette) manipulates an external *compiled help* file. You can display the contents, the index, or jump to a specific part of the help file (see Figure 17.21).

Figure 17.21. Control Online Help

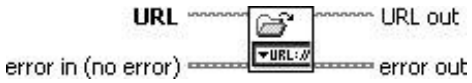


You can also distribute help documentation as plain HTML files, either along with your application or

on a web site at a specific URL. Use the Open URL in Browser.vi to open HTML files on disk by wiring a path data type to the URL input (this is a *polymorphic* VI that allows the URL to be passed as either a string or a path) or open a web page on the Internet by wiring a string containing the web page URL.

Open URL in Browser.vi (Programming>>Dialog & User Interface>>Help palette) displays a URL or HTML file in the default web browser. If the URL or path you wire to this VI contains a space character, the VI encodes the space as `%20` before displaying the URL or HTML file in the web browser (see Figure 17.22).

Figure 17.22. Open URL in Browser.vi



PREV NEXT

Pointers and Recommendations for a "Wow!" Graphical Interface

It's the attention to detail that often makes a graphical interface so startling that people go, "Wow!" To achieve this, we've collected a "bag o' tricks" over the years from experience and from observing other people's fancy VIs. Many VIs start out as a scratchpad with no concern for order or aesthetics, and never get cleaned up later. Believe me, it is worth the effort; even if you're not out to impress anybody, you and others will have an easier time working with your own VI. What if you have to modify that graphical spaghetti you threw together last year?

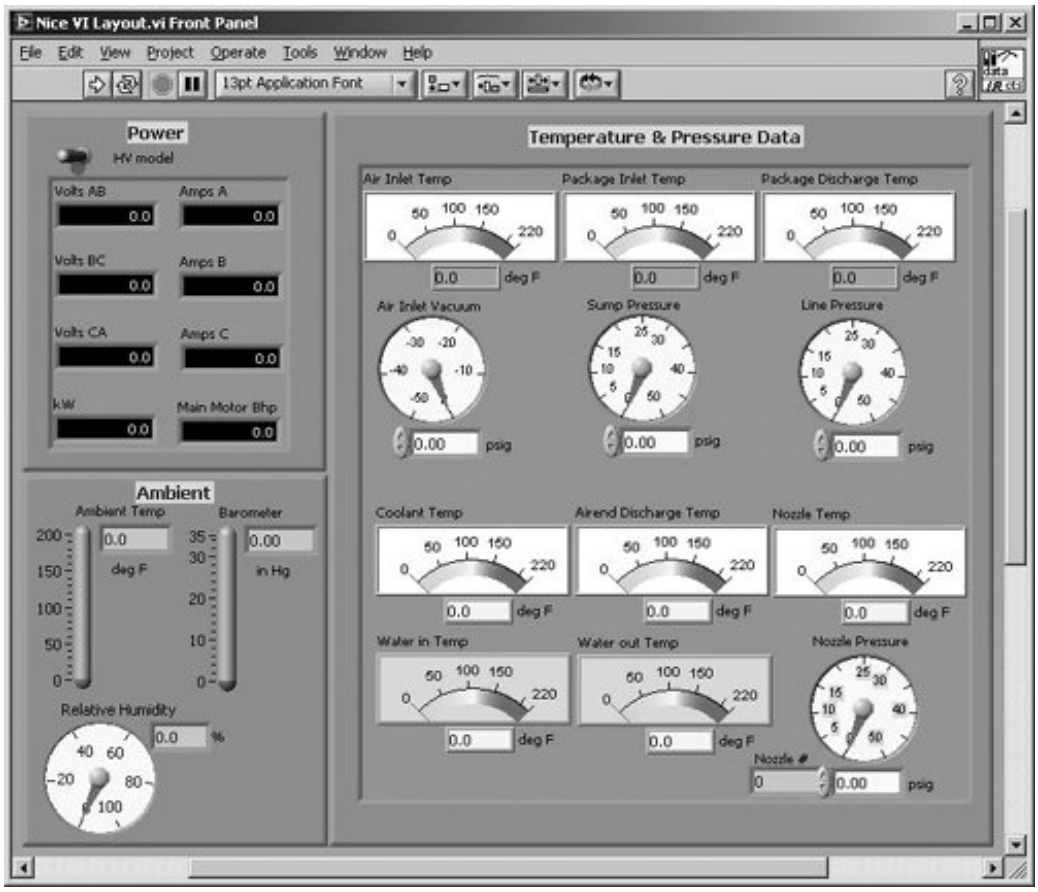
The following list is just a set of recommendations for a cool GUI; they don't necessarily always apply to your particular situationperhaps you'll find better onesbut they should help you get started.

Panel Layout

- If possible, make your top-level front panel and window size such that it fills up the whole screen, as long as you know that's the monitor size and resolution it will be used on.
- Try to use neatly aligned rectangular decorations as "modules" on the front panel, neatly aligned to group objects (see Figure 17.23).

Figure 17.23. A VI with a good modular layout of controls

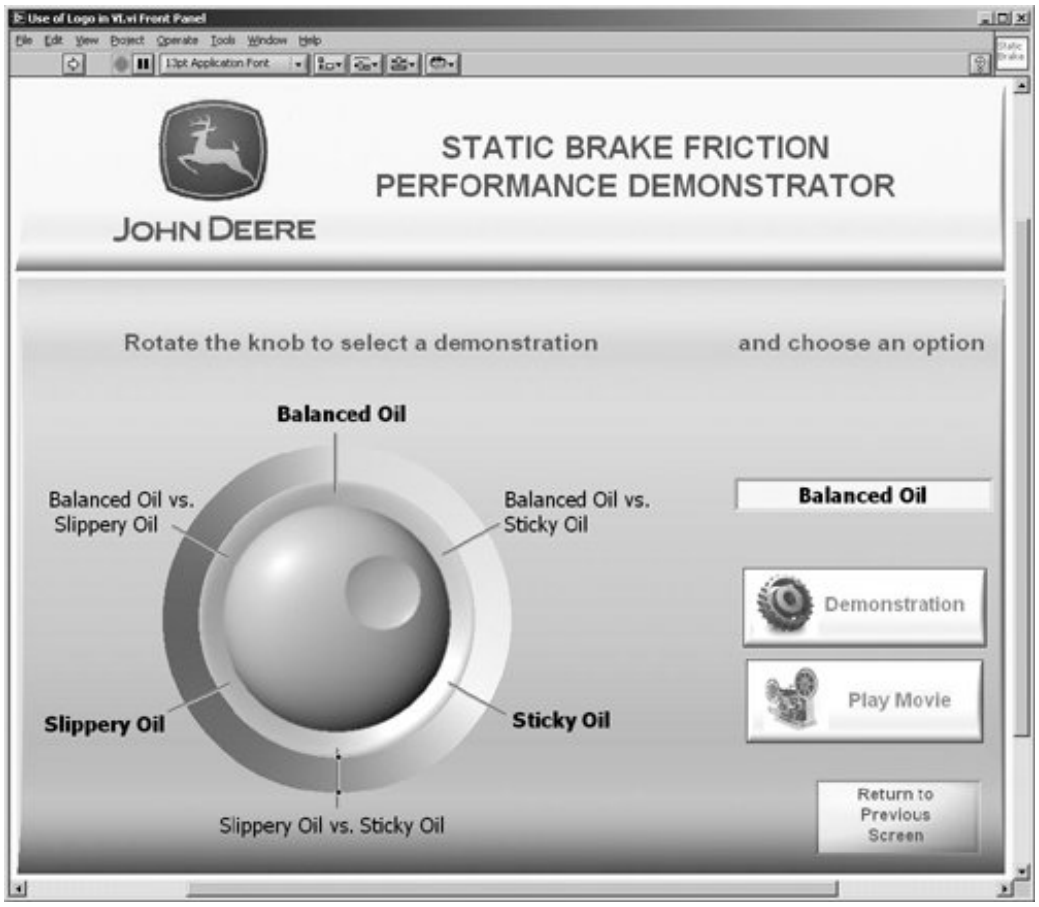
[View full size image]



- If you have empty space, consider filling it up with a decoration "module" as something you could use later; or fill it up with your company's logo or some cool picture. "Consider" it but don't cram the panel with meaningless stuff; some "white space" is good. (For example, there's a lot of judicious white space in the VI front panel shown in Figure 17.24.)

Figure 17.24. Use your company logo and its color scheme to work with your VI GUI.

[View full size image]



Text, Fonts, and Color

- Use color, but don't overdo it and be *consistent* with a coloring scheme that is maintained throughout the entire application. If you use red for a "hi" alarm in one panel, use the same red for the "hi" alarms in other panels (and please, no puke green or other weird colors . . .).
- Choose a background color different from your decoration box colors. Often a darker gray or black background adds a nice border effect. Alternatively, if you don't use the decoration modules, you may find that white or a soft, light color works best for the background.
- Select the "transparent" option for object label boxes from the Options... command under the Tools menu. Labels generally look much better without their gray box, as shown in Figure 17.25.

Figure 17.25. Using transparent label background to make the text more readable

- For many labels in general, and especially for the display text of numeric indicators and controls, coloring the background of the text black and making the text a vivid green or yellow makes them stand out. Choosing bold for the text usually helps as well.
- Use bright, highlighted colors only for items that are very important, such as alarms or error notifications.
- LabVIEW has three standard fonts (*Application*, *System*, and *Dialog* font). In general, stick to these if you are writing a cross-platform application. A custom font style or size you selected on a Windows machine may look very different (usually too big or too small) on Mac and Linux (and vice versa).

Graphs and Imported Pictures

- Make your panel look truly customized by importing your company's logo picture somewhere onto the front panel.
- Be aware that pictures and graph indicators can significantly slow down the update of your VI, especially if you have placed any controls or indicators on top of a picture or graph indicator.
- In general, you will want to select the Use Smooth Updates option from the Options . . . command to avoid the nasty "flickering" that would otherwise appear on graphs and other image-intensive operations.

Miscellaneous

- Include units on numerical controls if appropriate. Remember, LabVIEW has built-in units: If you choose to use these, you can show the unit label for numeric control or indicator by popping up on it and selecting Visible Items>> Unit Label. See Chapter 15, "Advanced LabVIEW Features," for a discussion of units. This allows the *end user* to quickly change the units of a control or indicator at *run-time* without you having to write any fancy code for converting the display units. They simply use the mouse to select the text in the Unit Label of the control or indicator, and then use the keyboard to type in the new unitsLabVIEW will validate that the typed units are of the required base units. (End users really love this featureespecially scientists and engineers.)
- Include custom controls when you can for neat visual effects.
- If you want to get really creative, for multimedia-type effects, use sounds and property nodes such as the position property.
- Use property nodes to hide or "gray out" controls when you want to indicate that these controls shouldn't be used.

- The VI Properties options, discussed in Chapter 15, are very useful for making your front panel fill the screen, center itself, and so on.
- Never forget to spend enough time on the icon editor, discussed in Chapter 5, "Yet More Foundations"make effective icons! Use a mixture of text and glyphs that are intuitive, professional, and easy to read (see Figure 17.26). They not only look better, but also make your code much easier to read and debug, especially if you are looking at a VI Hierarchy window. (On Windows, use 10-point [or 8-point all UPPERCASE] small font for easy-to-read, compact text.)

Figure 17.26. Using glyphs in VI icons helps you recognize them.

- When you see a good front panel design, learn and imitate!

If you're interested in learning more about LabVIEW GUI design, check out LabVIEW GUI: Essential Techniques by David J. Ritter (2002, McGraw Hill).

Some Final Words on the GUI...

Most of you at some time or another have probably had to give a presentation to your manager or some other audience about how your software works. The success of this "dog and pony show," in my experience, usually depends far more on the razzle and dazzle of the GUI than on whether the software is efficient or even works properly. Perhaps it shouldn't be this way, but that's often the way projects get accepted or rejected. The story goes that, at one large semiconductor manufacturer, some engineers had recently decided to revise a piece of software written in C that had taken two years to develop. Another engineer put together a LabVIEW front panel that demonstrated what the new software should look like. One of the top executives in the company was so impressed by the GUI that he immediately approved the engineer's proposal for all their worldwide plantsdespite the fact that the front panel was just a demo and didn't do anything yet!

The moral of the story is this: Spend some time making the best, coolest graphical interface you can. You never know how far it might carry you.

PREV NEXT

How Do You Do That in LabVIEW?

This section is more about programming solutions than about LabVIEW features. Common problems arise in developing many applications, and we show you some of them along with their solutions in the following pages. Some of the solutions are very simple; some are quite ingenious. By no means do all or even most programming challenges appear here; rather, these examples have been collected from a variety of sources to give you a jump start on some applications or at least food for thought.

You might take some of these problems as a challengetry to come up with a programming solution on your own first before looking at the answer!

There is an online list of LabVIEW frequently asked questions (FAQ) called the "LabVIEW FAQ" located at http://labviewfaq.org. The LabVIEW FAQ is hosted by the LabVIEW Advanced Virtual Architects (LAVA) user group, whose web site is located at http://lavag.org. See Appendix E, "Resources for LabVIEW," for more information about LAVA and the various LabVIEW resources they provide.

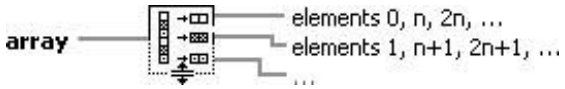
How Do You Acquire Data from Different Channels at Different Sampling Rates?

Things work fine when you're sampling all your data channels at the same rate. It's more challenging when one channel has to be sampled at 10 kHz, and another channel only requires sampling at 50 Hz. There's not really any magical solution to the multiple-sampling rate requirements, but you have a couple of options:

1. If you have more than one DAQ board, try assigning the channels so each board can handle the channels with the same sampling rate.

2. Sample everything at the highest rate, and then throw away the extra points on the lower-rate channels. One efficient way to do this is to use the Decimate 1D Array function (Programming>>Array palette).

Figure 17.27. Decimate ID Array



There are a few DAQ boards from NI that support simultaneous sampling with multiple ADC chips. In this case, you can set separate sampling rates for those channels with individual ADCs.

How Do I Clear a Chart Programmatically?

To do this, wire an empty array to the History Data attribute of a chart. The data type of this empty array is the same as the data type wired to the chart.

As a tip, don't do this when the VI starts, but instead do it when the VI exits. That way, when the VI loads, it will be ready to go and the user won't see any data from the last run at all.

There is an excellent example that illustrates this technique in `examples\General\Graphs\Charts.llb\How to Clear Charts and Graphs.vi`. The front panel and block diagram of this VI are shown in Figures 17.28 and 17.29, respectively.

Figure 17.28. How to Clear Charts and Graphs.vi front panel

[View full size image]

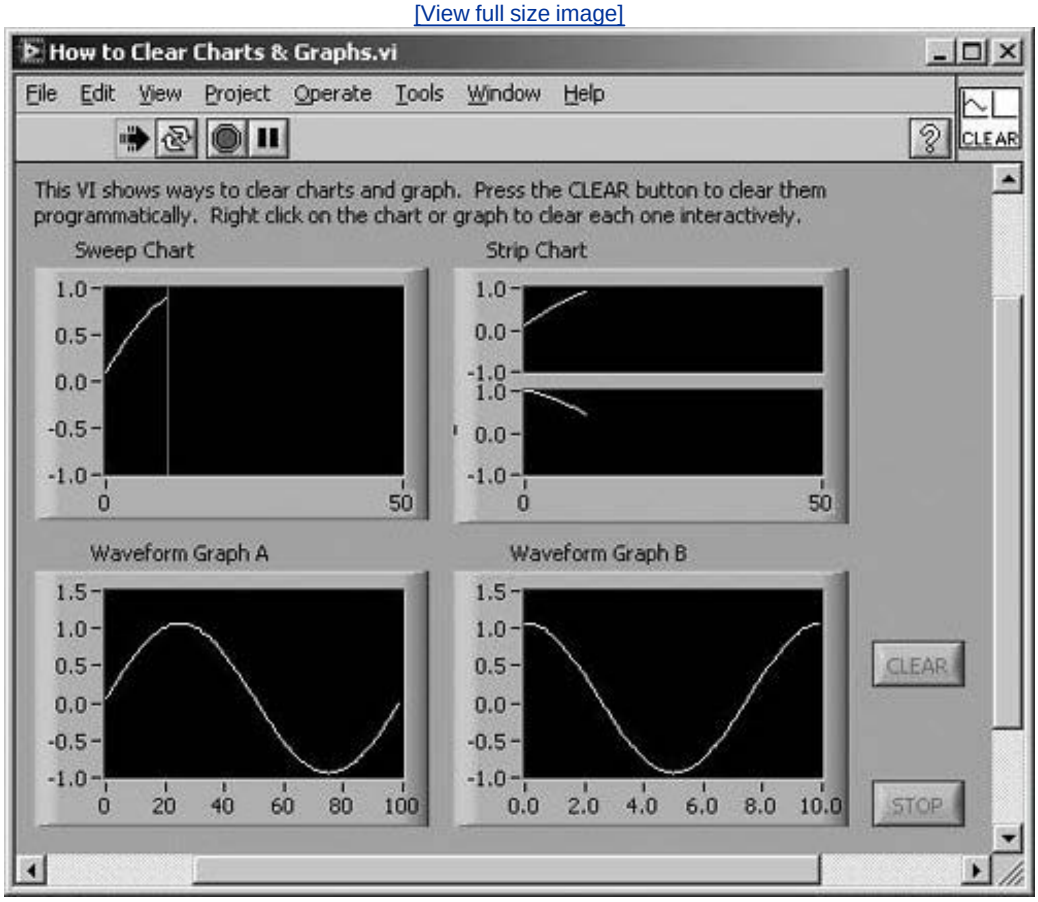
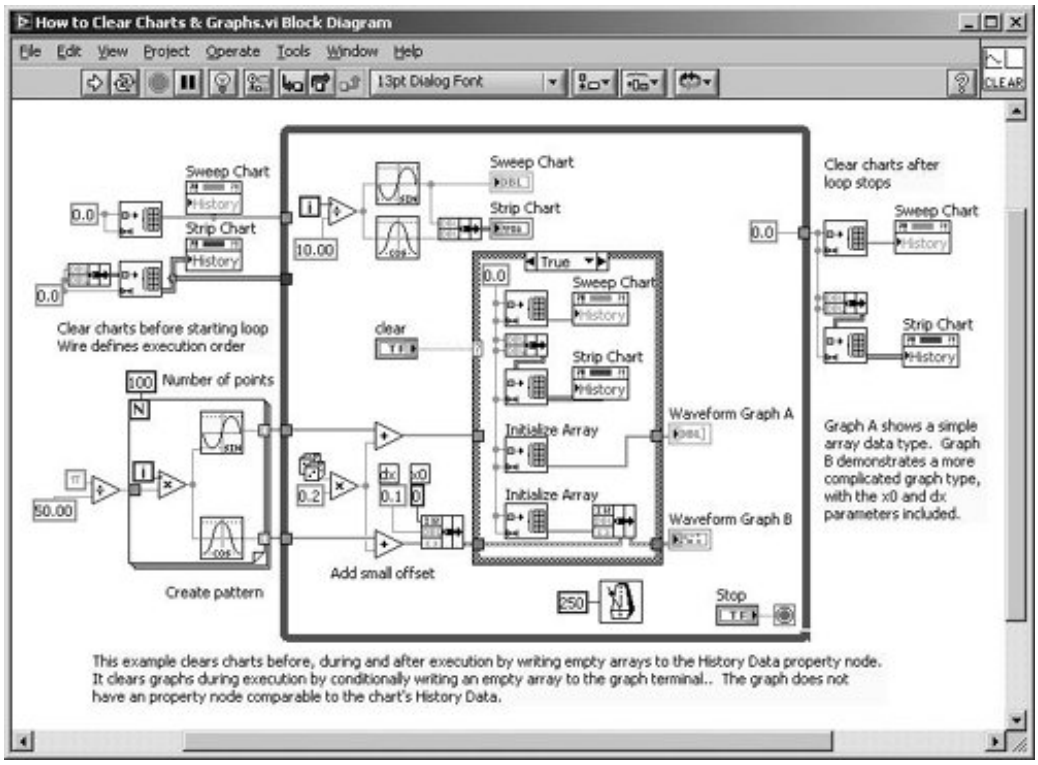


Figure 17.29. How to Clear Charts and Graphs.vi block diagram

[View full size image]

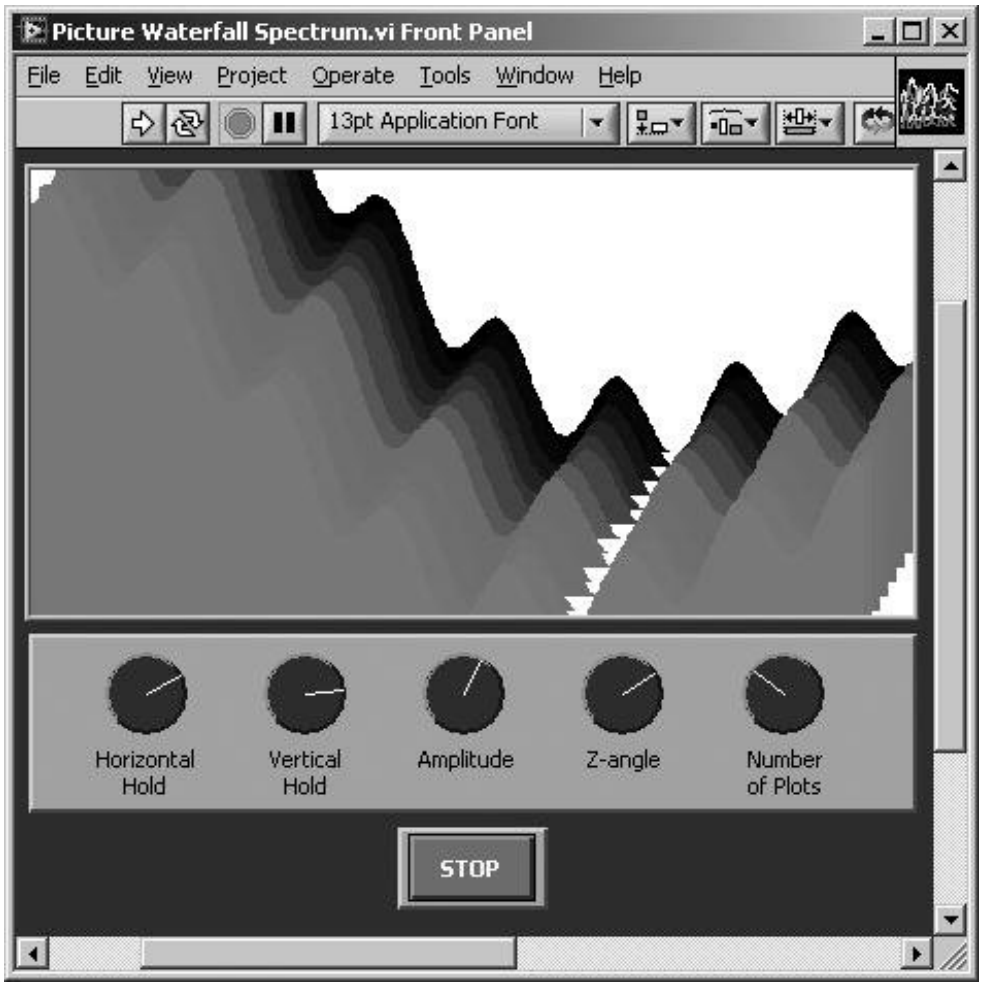


You can also pop up on the chart in the front panel and select Clear Chart.

How Can I Display a Waterfall Plot?

A waterfall plot is similar to a 3D plot, but places successive series of 2D graphs "stacked" along the Z-axis to show their change over time. In the LabVIEW examples, you will find "Picture Waterfall Spectrum.vi," which is shown in Figure 17.30.

Figure 17.30. Picture Waterfall Spectrum.vi front panel



This VI uses a Picture Control, which we discussed in Chapter 13, "Advanced LabVIEW Structures and Functions."

Is There a Way to Put More Than One Y Axis on a Graph for Two or More Plots?

Yes. On the graph, pop up and select Visible Items>>Scale Legend. Then resize the Scale Legend to add more *Y* axes. Another way to add a *Y* scale is to pop up on an existing *Y* scale and select Duplicate Scale from the shortcut menu.

My Graphs Flicker Every Time They Are Updated. It's Getting Really Annoying. Is There Any Way to Fix This?

Fortunately, yes. From the Tools menu, choose Options . . . . Select the Front Panel item. Check the box that says "Use smooth updates during drawing." This will turn off the flicker; the trade-off is

more memory will be used.

How Can You Create a Toolbar-Type Feature That Will Pop Up Different Sets of Controls and Indicators on the Same Window?

This was a difficult task to accomplish in earlier versions of LabVIEW. Now, however, you can use the SubPanel control to create a "frame" in your VI's front panel. The SubPanel is in the Modern>>Containers palette. It allows you to programmatically load VIs into this frame. You simply pass the VI Reference (see Chapter 15) of the VI you want to display in the SubPanel to the SubPanel's invoke node.

Can I Access the Parallel Port in LabVIEW?

Yes. Use the VISA Read and Write functions (never mind that parallel is the opposite of serial). Just as you use port 0 for COM1, port 1 for COM2, etc., in LabVIEW for Windows, port 10 is LPT1, port 11 is LPT2, and so on. You can even send data to a printer connected to a parallel port, use VISA Write, although you may need to know the printer's special control characters. Another good use for accessing the parallel port is to do some digital I/O without a plug-in board: You get eight digital lines for free! (But a hardware buffer is recommended to protect your computer.)

Also, you can use In Port.vi and Out Port.vi (found on the Connectivity>>Port I/O palette). These VIs allows you to peek and poke at the parallel port registers.

OpenG.org also offers a portIO library that allows reading and writing to memory addresses in Windows, which offers considerable performance *improvements over the VIs that ship with LabVIEW. See Appendix C, "Open Source Tools for LabVIEW: OpenG," for more information.*

Why Doesn't My VISA Session Work?

Did you open VISA sessions before trying to read and write? Did you get any error messages when you opened them? Did you supply the correct session IDs to the read/write VIs? You might want to report exactly what the error message is and exactly when it is reported. Cycle instrument and PC power, then use probes or simple error handler VIs to determine where the error first appears. Try sending an example message from the instrument manual verbatim. Verify that you are terminating your messages properly. Send your message to a front panel variable with "\" codes enabled to see

that you are sending exactly what you expect.

Also, try using the Instrument Assistant Express VI (which you learned about in Chapter 12, "Instrument Control in LabVIEW") to test communication with the device.

Can I Do Object-Oriented Programming in LabVIEW?

Yes. See Appendix D, "LabVIEW Object-Oriented Programming," for more information.

Can I Turn My VIs into Standalone Executables So People Can Run Them Without LabVIEW?

Yes. You need to either have the Professional version of LabVIEW or purchase the LabVIEW Application Builder add-on. Choose Build Executable . . . from the Tools menu to do this.

How Do I Monitor or Control My VIs over the Web?

Use LabVIEW's built-in web server to view or control your VIs. See Chapter 16, "Connectivity in LabVIEW," for more information.

Can LabVIEW Cook Me Dinner?

What are you hungry for? Don't assume "LabVIEW can't..." until you check out the documentation; you'll be surprised at how flexible and powerful LabVIEW is.

PREV NEXT

Memory, Performance, and All That

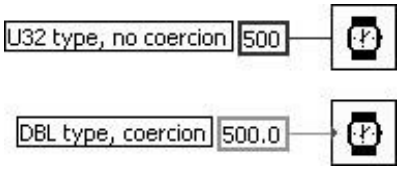
Becoming a better LabVIEW programmer means you know how to make applications that are mean and lean. Sure, if you have oodles of RAM and a dual terahertz processor, you may not need to worry about these things very much. But on average computers, critical or real-time applications are going to work better if you follow some simple guidelines for increasing performance and reducing memory consumption. Even if you don't foresee your application needing to conserve memory and processor speed, programmers who never take these issues into account arewell, just plain sloppy. Nobody likes debugging sloppy programs.

Curing Amnesia and Slothfulness

Let's face itLabVIEW does tend to make applications gobble memory, but you can make the best of it by knowing some tips. Memory management is generally an advanced topic, but quickly becomes a concern when you have large arrays and/or critical timing problems. Read anything you can find about improving LabVIEW performance and saving memory. Here is a summary of the tips:

- Are you using the proper data types? Extended precision (EXT) floats are fine where the highest accuracy is needed, but they waste memory if a smaller type will do the job. This is especially important where large arrays are involved.
- Globals use a significant amount of memory and time. Minimize not only the creation of globals, but the amount of times and number of places you read or write to them.
- Don't use complicated, hierarchical data types (such as an array of clusters of arrays) if you need more memory efficiency and speed.
- Avoid unnecessary coercion (the gray dots on terminals). Coercion indicates the expected data type is different from the data type wired to the terminalLabVIEW does an astounding job of accepting the data anyway in most cases (polymorphism), but the result is a loss of speed and increased memory usage because copies of the data must be made. This is especially true of large arrays.

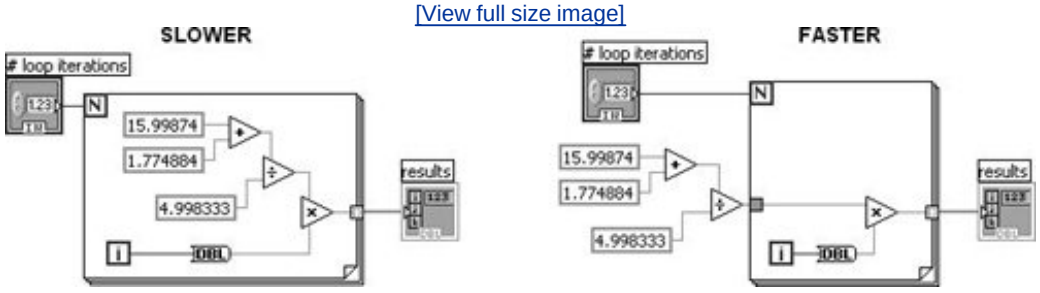
Figure 17.31. Numeric coercion



- How are you handling arrays and strings? Are you using loops where you don't have to? Sometimes there is a built-in function that can do the job, or several loops can be combined into one. LabVIEW is pretty smart about compiling efficient code, but it's still good practice to watch out for putting unnecessary elements into loops, as shown in Figure 17.32.

Figure 17.32. Putting computational work outside of loops, when possible, reduces redundant execution of code.

[View full size image]



- Where possible, avoid using Build Array inside loops, thus avoiding repetitive calls to the Memory Manager. Every time you call the Build Array function, a new space is allocated in memory for the whole "new" array (see Figure 17.33). Use auto-indexing or Replace Array Element with a pre-sized array instead. When auto-indexing in a While Loop, LabVIEW allocates an increasingly bigger chunk of memory than it needs each time it runs out, reducing the number of memory manager calls (see Figure 17.34). In a For Loop, it can calculate and allocate enough memory for all auto-indexed tunnels before it starts iterating (because a For Loop knows how many times it will iterate before it starts) (see Figure 17.35). Or you can do the calculation when you're writing the code, initialize an array to the maximum size or larger than the maximum expected size, and use the Replace Array Subset or Replace Array Element function instead of the Build Array (not shown).

Figure 17.33. Constructing an array inside of a While Loop using Build Array is very slow.

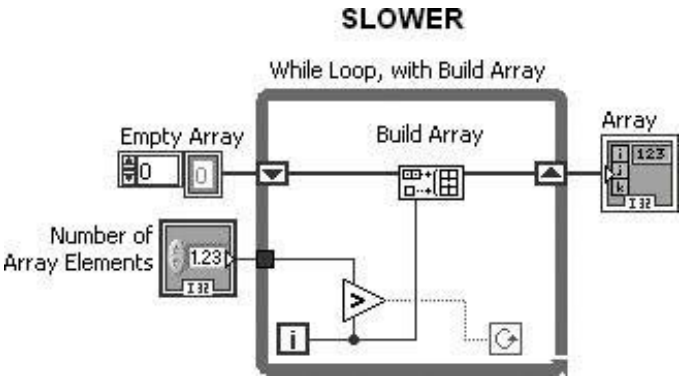


Figure 17.34. Using an auto-indexing tunnel is faster than using Build Array inside a While Loop.

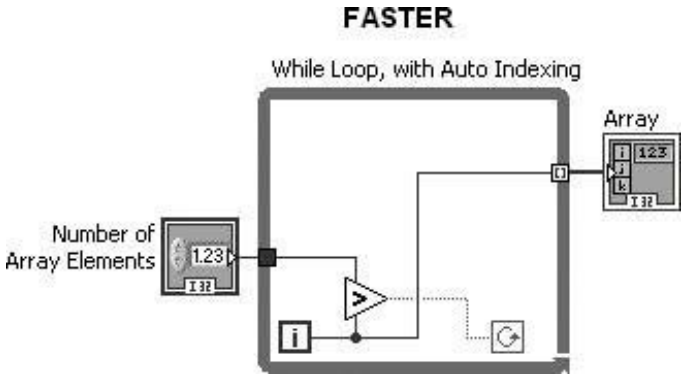
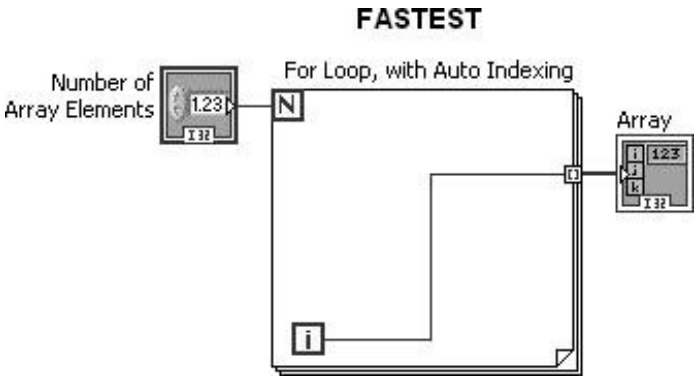


Figure 17.35. Using a For Loop with auto-indexing is faster than a While Loop with auto-indexing.



- Similar problems occur with Concatenate Strings. Unfortunately, LabVIEW usually can't make very good guesses about the lengths of strings you might choose to concatenate. If you have a decent idea at programming time, initialize a string larger than the maximum size ever expected (allocate an array of U8s and convert it to a string) and use the Replace Substring function (and a final String Subset, to trim off the excess) instead of the Concatenate Strings.

- Consider indicator overhead. Minimize the use of graphics (especially graphs and charts) when speed is extremely important.

- Update controls and indicators outside of loops whenever possible; that is, whenever it is not critical to view the object's value until the loop is finished executing.

- In the *Full* and *Professional* versions of LabVIEW, you can use the Show Buffer Allocations window (Tools>>Advanced>>Show Buffer Allocations menu) to see black squares on the block diagram that indicate where LabVIEW creates buffers to hold the data. Because you can see where memory is being allocated, you can try to reduce the memory required by minimizing the buffer allocations. Refer to the Fundamentals>>Managing Performance and Memory>>VI Memory Usage section of the LabVIEW Help documentation for information

about how LabVIEW allocates memory and for tips for using memory efficiently.

If you really want to peer into the memory consumption of your VIs, you can use the Profile Performance and Memory tool (from the Tools>>Profile>>Performance and Memory menu). You can also use the VI Metrics tool (from the Tools>>Profile>>VI Metrics menu) to see how many nodes, structures, etc. are in your VIs.

The Declaration of (Platform) Independence

LabVIEW does just an amazing job of porting VIs between different platforms. For the most part, you can take a VI created on any platform (Linux, Mac OS X, or Windows) and run it on another platform (as long they use the same LabVIEW version or a later one). However, not all parts of a block diagram are portable. If you want to design your VIs to be platform-independent, there are a few things you should know about portability:

- Be aware that the LabVIEW *application* and everything included with it (such as all the VIs in the `vi.lib` directory) are not portable. What the LabVIEW for each OS does is to *convert* VIs from other platforms, using its internal code, to VIs for its own platform.
- Some of the VIs in the Connectivity palette (like AppleEvents, ActiveX, and .NET) are system-specific and thus are not portable.
- VIs that contain CINs or Call Library Functions are not immediately portable. But if you write your source code to be platform-independent and recompile it on the new operating system, the CIN should work fine. When calling shared libraries, you can create a shared library for each platform that has the same file name, but a different file extension for each platform (`.dll` for Windows, `.framework` for Mac OS X, and `.so` for Linux). In the "browse" file dialog of the Call Library Function, replace the file extension with an asterisk "*," which will cause LabVIEW to look for a shared library that matches the current system's naming convention. This allows you to use your VIs on all platforms without editing the code. Just ship your VIs along with the platform-specific shared library.
- Keep in mind such things as filenames that have their own special rules for each OS, and don't use characters such as [`:/\`], which are path delimiters in different operating systems. Store paths constants as the path data type, rather than the string data type, and LabVIEW will automatically convert the path delimiters. For example, a path constant stored as "..\setup.ini" in Windows will be automatically converted to "../setup.ini" when the VI is run on Linux. If you store your paths as string constants, this conversion will not occur and your code will probably not run correctly.
- The end of line (EOL) character is different on each platform (`\r` for Mac OS X, `\r\n` for Windows, and `\n` for Linux). The easiest solution is to use LabVIEW's End of Line constant (), from the **String** palette.
- Fonts can be a real mess when porting between systems. If you can, stick to the three standard LabVIEW font schemes (Application, Dialog, and System) and don't change their sizes, because custom fonts that look fine on one platform may look huge, tiny, or distorted on another. Also, if the Boolean text of a button grows in size, it can cause the button size to grow, as well.

- You can use system controls, indicators, decorations, and colors (as described in Chapter 13) to create user interfaces that adapt to the style of the operating system and theme of the end user.

- Screen resolution can also be a nuisance. Some people recommend sticking to using a screen resolution of 1024 x 768, which will make VI windows fit fine on most monitors. Remember, keeping diagrams small means making your block diagrams organized and modularthis is always a good idea!

PREV NEXT

Programming with Style

Programming really is an art... and it can be especially fun in LabVIEW! The following section is a collection of final reminders and guidelines for writing a *good* LabVIEW application. You can look at the Fundamentals>>Development Guidelines> Concepts>>LabVIEW Style Checklist section of the LabVIEW Help documentation for a checklist of style and quality guidelines.

If you're interested in learning more about LabVIEW coding style, check out The LabVIEW Style Book *by Peter Blume (2006, Prentice Hall).*

One of the best ways to learn good style *is by looking at the LabVIEW code of developers who use good style. One good place to find such examples is on the block diagrams of OpenG VIs. These VIs are under heavy peer review and scrutiny and, as a result, have code that is very well organized and commented.*

Modularize and Test Your VIs

Although it's theoretically possible to be *too* modular in designing your program, this rarely happens. Make all but the most simple and trivial functions and procedures subVIs. This gives you a chance to test each of your individual pieces of code before working with the big piece. It also lets you easily re-use code, keep yourself organized, and make the block diagram size manageable. Don't forget to test each subVI as a top-level VIand be thorough: Test all sorts of strange input combinations. If you know all your subVIs are working, it should be very easy to debug any problems with your top-level VI.

One useful tip: Often LabVIEW programmers won't test certain VIs because they require DAQ hardware or other externally-generated inputs. Don't wait until you have the hardware. Write a simple "dummy data" VI (or better yet, use a simulated DAQmx device as we learned about in Chapter 10, "Signal Measurement and Generation: Data Acquisition") to pass data to these VIs in the meantime so you can at least test part of their functionality.

Document as You Go Along

Please document your work! Many programmers shun documentation, thinking it will be done later or not needed . . . until a user (possibly yourself) two years later is trying to figure out how this VI works. Take advantage of LabVIEW's built-in documentation capabilities:

1. Block diagram comments. Use floating labels (or edit function and structures labels) to comment your code. Don't get carried away, restating the obvious, but definitely place comments that give people an overview of what the code fragments are intended to do.
2. VI documentation. At the very least, write a short description for each VI you create. This is extremely valuable when someone is looking at the subVI icons in the block diagram and needs an idea of what they do.
3. Descriptions. Ideally, write a help statement for each control and indicator using the pop-up Description and Tip . . . command (unless you've chosen the control label or caption so well that it tells the whole storyvery unlikely). These invaluable comments will appear on the Help window if a user points to the control or indicator in question.
4. VI revision history. This option, available from the Edit menu, is a more sophisticated tool for larger projects. It allows you to enter comments about the changes you've made to a VI along the way. The History window can be quite helpful when more than one person works on a project, because it keeps track of the user, time, and date. (It also can be invaluable when someone manages to link an old copy of a subVI into your project, and old bugs that you know you've fixed reappear.)
5. *Front panel text*. For important indications, just write some text (perhaps with a bold or large font) on the front panel itself. Users can't miss that one!

One More Time: Dataflow!

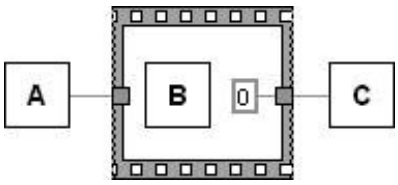
As you get more comfortable with LabVIEW, you will begin to take more advantage of the way dataflow programming works. Some tips to remember are the following:

- Dataflow means that data is carried across wires. When data reaches a terminal of a subVI or a function, that subVI or function will only start executing after ALL of its input terminals have received data.
- Two or more objects or groups of objects on the block diagram that are not connected have no specific execution sequence. Many people new to LabVIEW have a feeling that execution should

take place left-to-right or top-to-bottom. That is not true! There is no way to predict in what order two or more pieces of block diagram will occur unless it's specified by dataflow. (Unconnected nodes won't even necessarily execute in the same order each time, so execution highlighting doesn't help predict the order.)

- When you need to "force" an execution sequence, you have the option of using a sequence structure, which can be awkward for large diagrams, or an artificial data dependency structure, as shown in Figure 17.36. (And, yes, most experienced LabVIEW programmers use a single frame sequence for this purpose when there isn't a convenient terminal availablebecause a "sequence" is self-documenting.)

Figure 17.36. Even though B requires no inputs from A, by wiring the output of A to the sequence structure, the diagram forces B to execute only after A has completed executing. C similarly does not begin execution until the entire content of the sequence, including B, has finished.



- You've noticed that many LabVIEW functions have common "threads": refnums, taskID, error clusters, and so on. These threads are designed to string several common VIs together in a natural dataflow sequence. You can also use threads, such as error clusters, for your own VIs. Using threads will reduce the need for sequence structures with awkward sequence locals.

PREV NEXT

Wrap It Up!

This chapter gave you some instructions on the *art* of graphical programming. We looked first at the front panel end: suggestions, guidelines, and reasons for making an exciting graphical user interface (GUI). Second, we focused on the block diagramprogramming solutions, performance, memory, and style.

Creating a good GUI is important for "selling" your program to your customer or your boss, as well as making it much easier to use. LabVIEW's "art" enhancements include decoration modules, align and distribution commands, and layered objects.

Custom controls and indicators can add value to your GUI by providing graphical simulation and animation tools. The Control Editor lets you modify standard LabVIEW controls and indicators and import picture files to represent the new objects.

The Help window is not just for you, the programmer. The end user can use the Help window to examine front panel object's descriptions. You can open or close the Help window programmatically.

Many common questions and problems arise when you are programming in LabVIEW. Some of the solutions were discussed in this chapter. You can find more ideas to solve your specific problem by consulting the LabVIEW Frequently Asked Questions (FAQ) (http://labviewfaq.org) and the resources listed in Appendix E, "Resources for LabVIEW."

When you need to improve the speed and/or memory performance of LabVIEW, follow the guidelines discussed in this chapter. Watch out always for unnecessary operations inside loops, especially with arrays.

Although for the most part LabVIEW VIs are platform independent, you do need to be aware of a few obstacles that can pop upsuch as DLLs or system-specific functions.

Finally, to be a good LabVIEW programmer, you need to be systematic in making modular VIs that are tested thoroughly. Good documentation is essential to making a quality, maintainable piece of software. Last but not least, learn to use LabVIEW's distinctive hallmark unknown to other languagesdataflowto your advantage.

Concluding Remarks

This is the end of the book! (Well, there are some appendices, and if you're really bored, you can read the index.) By now, you've gained a solid understanding of how LabVIEW works. You've also begun to see how it can work for you, whether your application is teaching an electrical engineering class or building a process control system for a large plant.

Where do you go from here? More than anything else, hands-on experience is the best teacher. Experiment with a VI. Build a prototype. Look at examples. Be creative. Above all, have fun. If you decided to buy the full version of LabVIEW, don't be afraid to go through the manuals and built-in examples. They can be an invaluable reference for details on your application. Get involved with user groups like Info-LabVIEW (info-labview.net), LAVA (lavag.org), OpenG (openg.org), and the Developer Zone at ni.com. Finally, please check out our web sites at jeffreytravis.com, jameskring.com, and jkisoft.com.

Good luck and, as we like to say, Happy Wiring!

Jeffery Travis

Jim Kring

PREV NEXT

A. CD Contents

The following is a list of items found on the accompanying CD-ROM:

README.txt: Read this document first. It contains information about the CD-ROM.

index.html: This is an HTML file that contains descriptions and links (shortcuts) to all files on this CD-ROM. You can view this file using any web browser.

LABVIEW_80_INSTALLER folder: In this folder, you will find a 30-day evaluation version of LabVIEW 8.0 (Windows version only), which allows you to do just about everything the commercial version does during the evaluation period. You can always get the latest evaluation version of LabVIEW at http://ni.com/labview.

EVERYONE folder: This folder contains the example and activity LabVIEW VIs from the book. Each chapter's activities are in a corresponding subfolder; for example, you will find the activities for Chapter 9, "Exploring Strings and File I/O," in CH09.

CERTIFICATION folder: Here you will find information and sample exams related to the official Certified LabVIEW Associate Developer (CLAD), Certified LabVIEW Developer (CLD), and Certified LabVIEW Architect (CLA) examinations and certification process. See Appendix F, "LabVIEW Certification Exams," for more information on this topic.

For the updated files, errata, and more information, visit LabVIEW for Everyone *on the Web at http://labviewforeveryone.com.*

B. Add-on Toolkits for LabVIEW

You can purchase special add-on toolkits to increase LabVIEW's functionality. In addition, new toolkits are created frequently, so if you have a particular goal, it's worthwhile to check and see if a toolkit already exists to accomplish it. Some toolkits are sold by National Instruments; others are created by third-party companies (often referred to as "Alliance members").

You can also find free, open source toolkits for LabVIEW at OpenG.org (http://openg.org).

You can search for LabVIEW tools on the LabVIEW Zone Tools Network on the Web at http://ni.com/labviewtools/.

The following is a list of *commercial* toolkits that are available from National Instruments (http://ni.com). For more information on open source toolkits, see Appendix C, "Open Source Tools for LabVIEW: OpenG."

Application Deployment and Module Targeting

- Application Builder
- Remote Panels
- NI Motion Assistant
- NI SoftMotion Development Module for LabVIEW
- LabVIEW PDA Module
- LabVIEW Real-Time Module
- LabVIEW FPGA Module
- LabVIEW Vision Development Module
- Math Interface Toolkit

Software Engineering and Optimization Tools

- Execution Trace Toolkit for LabVIEW Real-Time
- Express VI Development Toolkit
- State Diagram Toolkit
- VI Analyzer Toolkit

Data Management and Visualization

- Report Generation for Microsoft Office
- Database Connectivity Toolkit
- Enterprise Connectivity Toolkit
- Internet Toolkit
- DIAdem
- NI INSIGHT for 3D Test and CAE Visualization

Real-Time and FPGA Deployment

- LabVIEW Real-Time Module
- PID Toolkit for Windows
- Execution Trace Toolkit for LabVIEW Real-Time
- LabVIEW FPGA Module

Embedded System Deployment

- DSP Test Integration Toolkit
- Embedded Test Integration Toolkits for LabVIEW
- Digital Filter Design Toolkit

Signal Processing and Analysis

- Sound and Vibration Toolkit
- Advanced Signal Processing Toolkit
- Modulation Toolkit
- Spectral Measurements Toolkit
- Order Analysis Toolkit
- Digital Filter Design Toolkit
- Math Interface Toolkit

Automated Test

- IVI Driver Toolkit for Windows
- NI TestStand
- Database Connectivity Toolkit
- Switch Executive
- NI Analog Waveform Editor
- NI Digital Waveform Editor

Image Acquisition and Machine Vision

- LabVIEW Vision Development Module
- NI Vision Builder for Automated Inspection
- NI-IMAQ for IEEE 1394
- LabVIEW FPGA Module

Control Design and Simulation

- Control Design and Simulation Bundle
- LabVIEW Real-Time Module
- System Identification Toolkit
- Control Design Toolkit
- LabVIEW Simulation Module
- Simulation Interface Toolkit
- State Diagram Toolkit

Industrial Control

- LabVIEW Datalogging and Supervisory Control Module
- PID Toolkit for Windows
- NI Motion Assistant
- NI SoftMotion Development Module for LabVIEW
- LabVIEW Vision Development Module
- VI Logger
- Lookout
- LabVIEW PDA Module
- LabVIEW Real-Time Module
- NI Industrial Automation OPC Servers
- Motion Control Tools

C. Open Source Tools for LabVIEW: OpenG

Free Open Source Software

OpenG.org: Home of the LabVIEW Open Source Community

Free Open Source Software

Just about everyone involved with software and computers has heard the term *open source*, or more specifically, *free and open source software* (FOSS). Most people have used FOSS, such as Mozilla Firefox and perhaps GNU/Linux. In fact, LabVIEW has some open source components such as the Mesa 3D open source cross-platform graphics library and the Perl Compatible Regular Expressions (PCRE) library, which powers its Match Regular Expression function.

OK, so what is FOSS? Generally speaking, FOSS is software that is licensed in a way that gives users the right to view, modify, and improve the source code. The term *open source*, on the other hand, is a much more specific term that generally refers to software that is licensed under terms that meet the criteria of the *open source definition* (http://opensource.org/docs/definition.php), which is maintained by the Open Source Initiative (http://opensource.org).

The open source definition defines the following set of criteria for being open source certified:

1. Free Redistribution
2. Source Code
3. Derived Works
4. Integrity of the Author's Source Code
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavor
7. Distribution of License
8. License Must Not Be Specific to a Product
9. License Must Not Restrict Other Software
10. License Must Be Technology-Neutral

These criteria are designed to protect (1) the author/owner of the code, (2) the user of the code, and (3) the code itself. There are a variety of open source-certified licensesyou can find a list of them on the Web at http://www.opensource.org/licenses/and each of them places the priorities of protection for the three participants (author, user, and code) at differing levels based on the interests of the developers, sponsors, and users of the software.

OpenG.org: Home of the LabVIEW Open Source Community

OpenG.org is a community of LabVIEW developers that are committed to the development and use of open source tools for LabVIEW. They produce several libraries, such as an application build tool, package management tool, installer builder, and a plethora of reuse libraries that are collectively referred to as the OpenG Toolkit. You can find more information about OpenG, as well as download the tools, at http://openg.org (the OpenG.org web site is shown in Figure C.1).

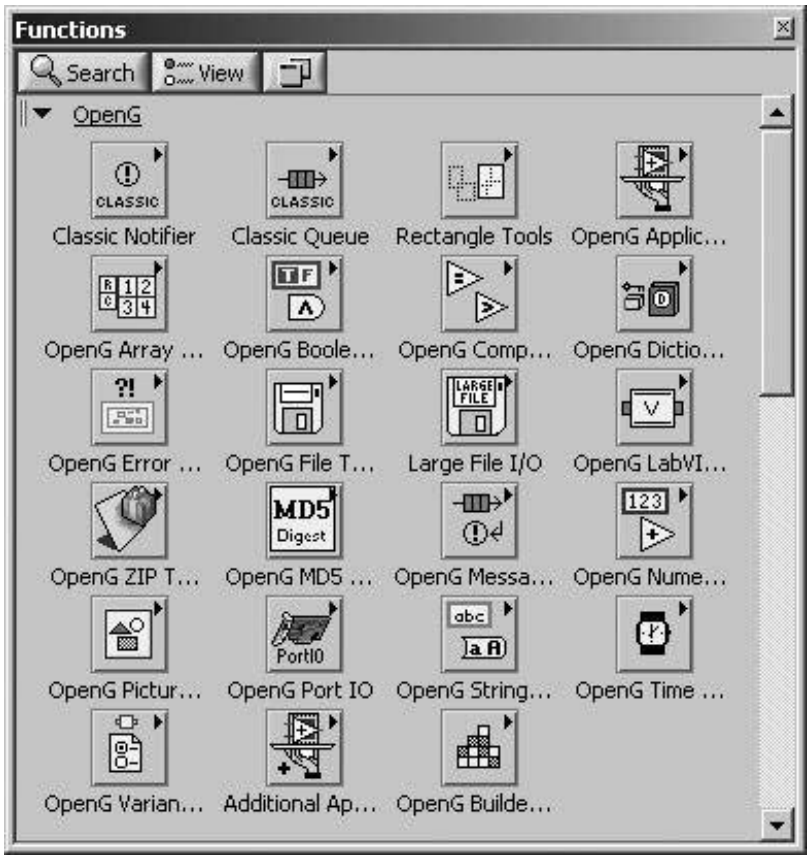
Figure C.1. The OpenG.org web site

[View full size image]

Installing the OpenG Tools is very easysimply download and install OpenG Commander (click on the "Get OpenG Commander" button on the OpenG.org home page) and choose which packages you wish to install. OpenG Commander will download them directly from the Internet and install them directly into your Functions palette!

Figure C.2 shows some of the OpenG tools installed in the Functions palette.

Figure C.2. Some of the OpenG libraries installed in the palettes



OpenG Commander will also let you know when there are new versions of packages available. Sounds great, doesn't it? And, the best part about OpenG.org is that all the tools are created by people *just like you!* They are sharing and collaborating to help make programming LabVIEW applications easy and fun, and isn't that the point? If you are interested in participating in OpenG.org, visit their web site (http://openg.org) and join in on the discussion forumsthey're all friendly and talented LabVIEW developers who enjoy working with others.

It's also worth mentioning that some features in LabVIEW are inspired by work done by OpenG. So, participation in LabVIEW is another way that you can help make LabVIEW a better programming language!

PREV NEXT

D. LabVIEW Object-Oriented Programming

Introduction to Object-Oriented Programming

If you have programmed in other languages, you have probably heard of (and maybe have used) object-oriented programming (OOP) techniques in some of your previous applications.

This appendix assumes you are somewhat familiar with concepts of object-oriented programming, and that you have mastered the basics of LabVIEW. If not, much of the material here will probably confuse you. Don't feel badOOP is a very big and advanced topic. Learn about OOP and learn LabVIEW, though, and you will have some very good programming skills to work with.

OOP is a (really) big subject, and there are a lot of good resources for learning OOP. We encourage you to spend some time learning about OOP, in order to better understand LabVIEW object-oriented programming techniques. You can find a lot of good references simply by googling "object-oriented programming."

Object-Oriented Programming Concepts

The following concepts are central to OOP:

- Class The definition of the data type and behavior of objects.
- Object An instance of a class.
- Encapsulation The ability of objects to hide specific details of how they implement their behavior.
- Inheritance The ability of classes to specialize another class (without having to modify the class that is being specialized).
- Abstraction The ignoring of specific characteristics of objects that are not important to the work at hand.
- Polymorphism The ability of different classes to each implement a behavior in their own way. The definition of how a behavior is implemented is defined inside the class.

Programming languages that support all of these concepts, in the editor and compiler, are referred to as *object-oriented languages*. Many programming languages support some of these concepts, including LabVIEW. And, LabVIEW continues to become more of an object-oriented language, even as this book is being written.

The objective (no pun intended) of OOP is to design systems that can evolve with as little (human) effort as possible. OOP strives to minimize the amount of work required to make incremental changes and improvements to the system. This is generally achieved through loose couplingavoiding unnecessary dependencies that prevent software components from being modular, maintainable, and reusable.

One word of caution about object-oriented programmingit's not the best tool for every problem. Do use OOP to implement an object-oriented model of your system. Don't use OOP to solve every programming task.

Object-Oriented Analysis and Design

Before we discuss more about LabVIEW OOP, it is important to discuss object-oriented analysis (OOA) and object-oriented design (OOD). OOP is simply a tool that allows you to implement models that are developed using object-oriented design techniques. Without a good design, OOP is no better than any programming tool applied to a poorly designed system. And, models should be designed such that they reflect real-world systems (because we are trying to automate the real world). The process of analyzing real-world systems in order to create object-oriented designs is referred to as object-oriented analysis. There are some very good (and standardized) methodologies and techniques for doing object-oriented analysis and design. Again, you can find some of these by googling "object-oriented analysis" and "object-oriented design."

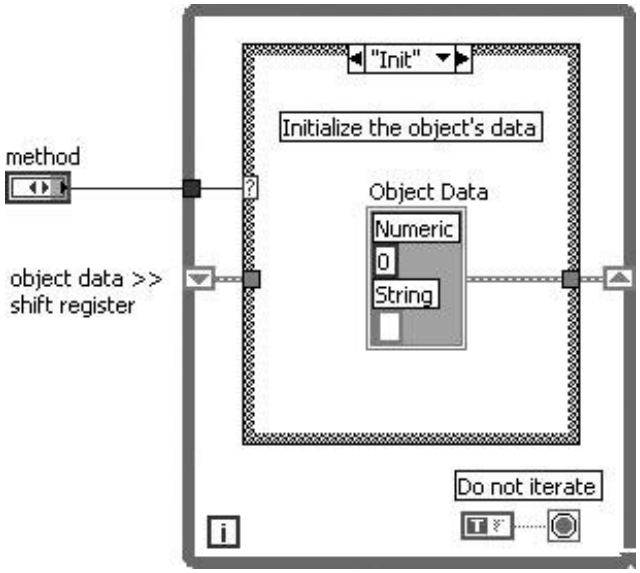
PREV NEXT

LabVIEW Object-Oriented Programming

Functional Globals

A *functional global* is a LabVIEW programming construct (or pattern) that is very commonly used. It is a simple and elegant way to do object-oriented programming in LabVIEW. A functional global (which acts as our *object*) is a VI, which contains a While Loop + case structure combination, as shown in Figure D.1

Figure D.1. The essential parts of a functional global



The While Loop is configured to iterate only once. (Note that the return selector is configured to Stop if True and a value of TRUE is wired to it.) The While Loop contains a shift register that is used exclusively for storing the object's *data* (a shift register is much faster than other storage mechanisms, such as globals and locals, because there is only one copy, ever, of the data in memory). Each frame of the case structure is a *method* that may be called by setting the enum value that is passed into the functional global. The first case of the case structure is the "Init" frame that defines the object's data type.

It is important that functional global VIs are not reentrant, so that each location where it is called as a subVI refers to the same instance of the VI. This allows all subVI instances to refer to the same object (they all share the same shift register "object" data). If, however, the functional global were reentrant, each instance would have a unique and independent data space (having its own independent copy of the shift register data). Refer to Chapter 15, "Advanced LabVIEW Features," for more information about the reentrant VI setting, its features, and considerations.

Make sure that the "method" enum is a type definition (as discussed in Chapter 13, "Advanced LabVIEW Structures and Functions") because you will be using this enum in many locations. You will want all instances of this enum to be updated when you modify (add, remove, rename, etc.) the methods of your object.

A functional global can be thought of as a *singleton*, an object-oriented pattern where only one instance (or object) of a class will exist. The functional global is both a class and an object, rolled into one.

Example: Functional Queue.vi

Now let's look at an example of how we will use the functional global, in practice. We will create a queue object that, in addition to the standard "Init" method, has an "enqueue" method for adding elements to the queue and a "dequeue" method for removing elements from the queue.

Functional Global "Core"

The functional global "core" is a single VI that contains the object data and methods. When programming applications that use a functional global object, you will not call the functional global "core" as a subVI directly; rather, you should create public method VIs, one for each method in your functional global "core." Keep this in mind while we discuss the functional global "core," as it will make more sense when the public methods are introduced in the next section.

Figure D.2 shows the front panel of a functional global core named Functional Queue.vi. The `method` enum has one element for each method, and each method (except for "Init") has an input and output cluster on the front panel. The `method` enum and the input/output clusters have all been made

type definitions, as described in Chapter 13.

Figure D.2. Functional Queue.vi front panel

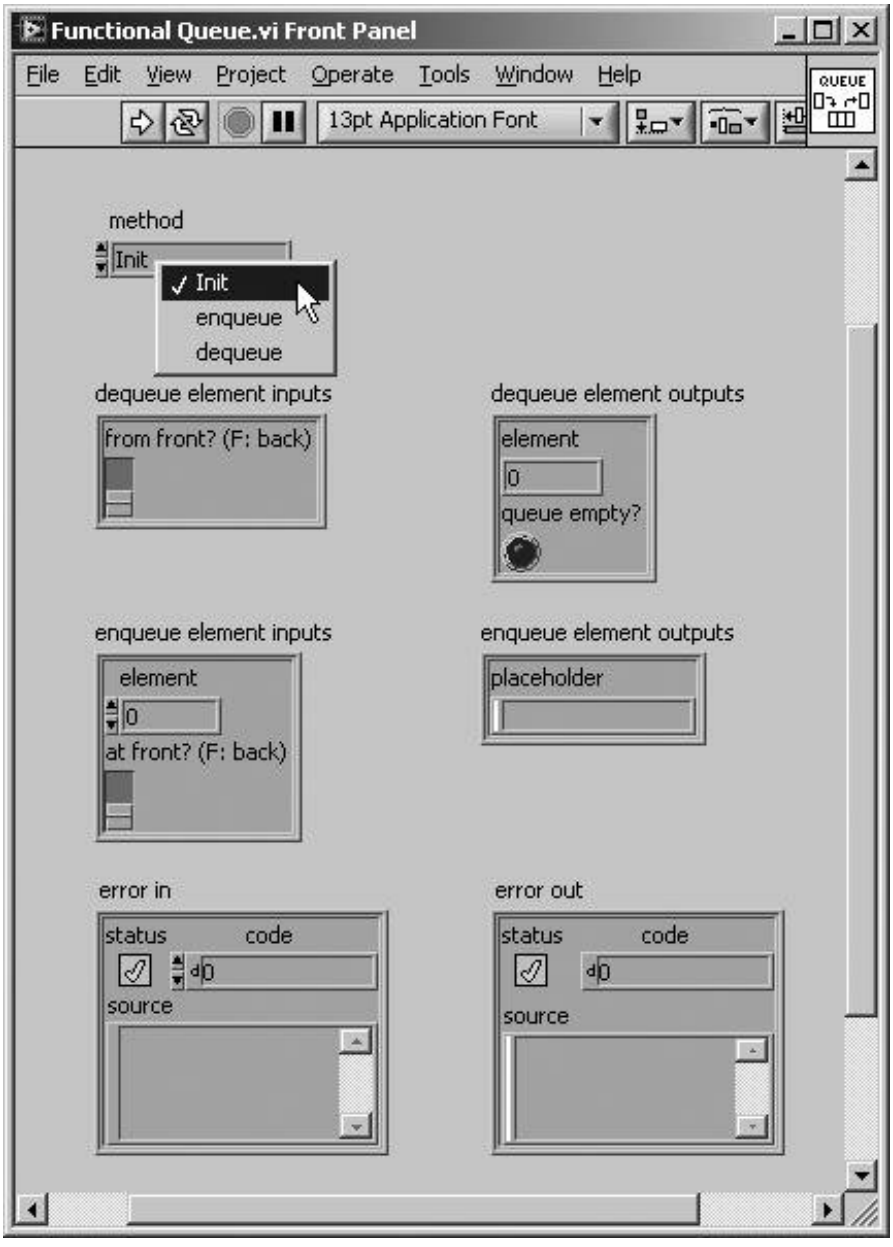
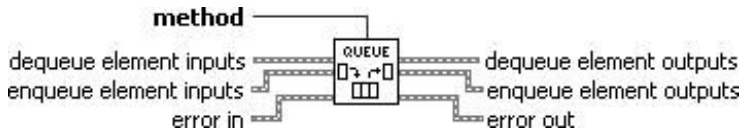


Figure D.3 shows the connector panel of Functional Queue.vi. Note again that there is one input cluster and one output cluster for each method declared in the `method` enum.

Figure D.3. Functional Queue.vi connector pane



Now let's take a look at the block diagram of Functional Queue.vi. Figure D.4 through Figure D.6 show the block diagram of Functional Queue.vi with the three method case frames visible: "Init," "enqueue," and "dequeue." Note that each frame unbundles only its own input cluster and bundles only its own output cluster. Each method accesses only its own input arguments and passes out only its output arguments.

Figure D.4. Functional Queue.vi's "Init" method case

[View full size image]

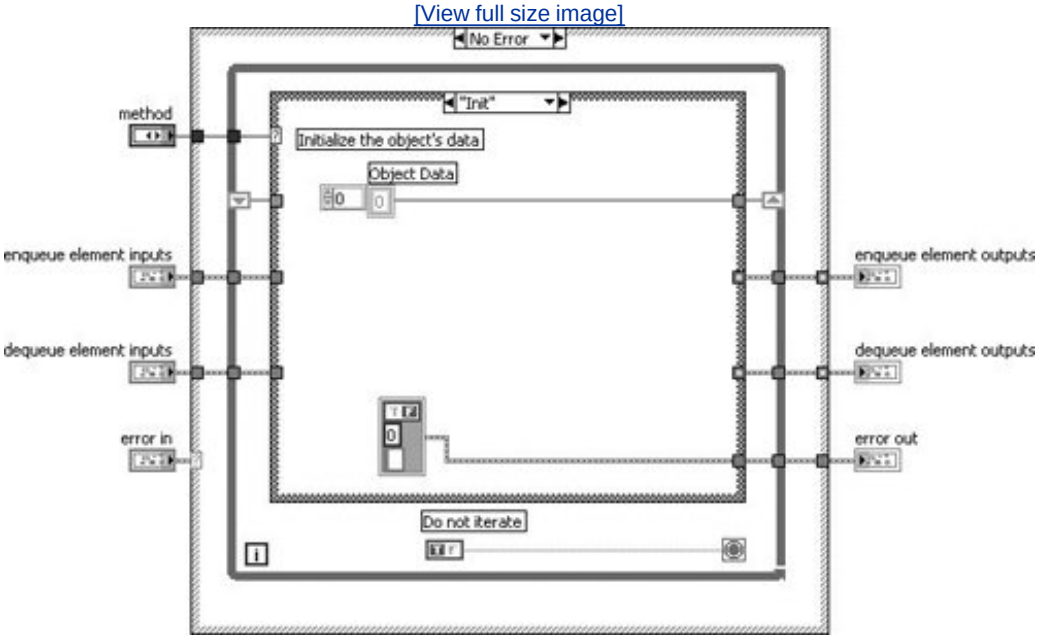


Figure D.5. Functional Queue.vi's "enqueue" method case

[View full size image]

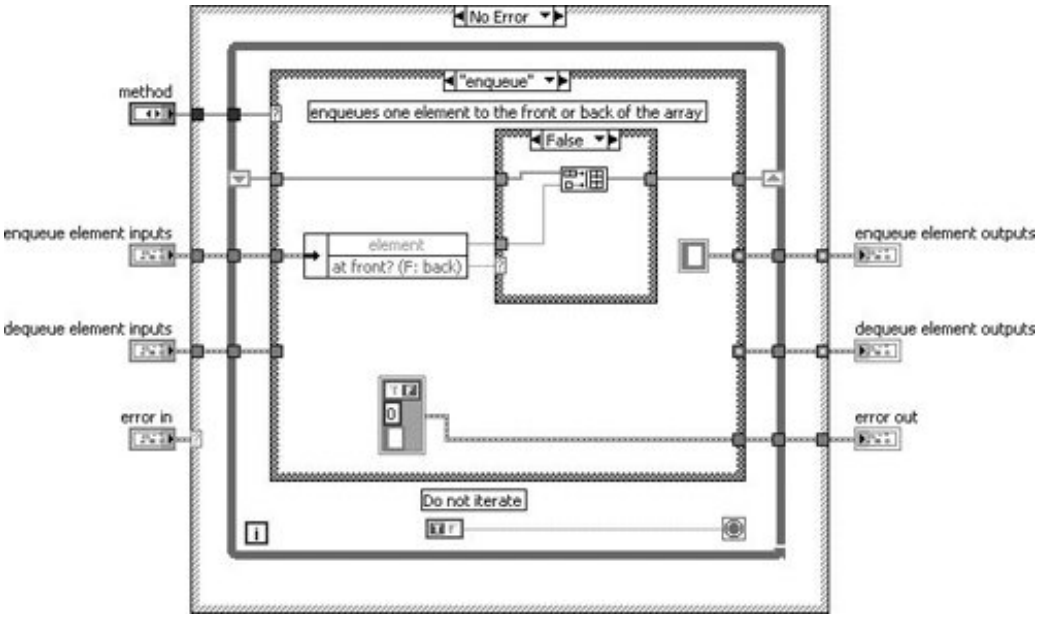
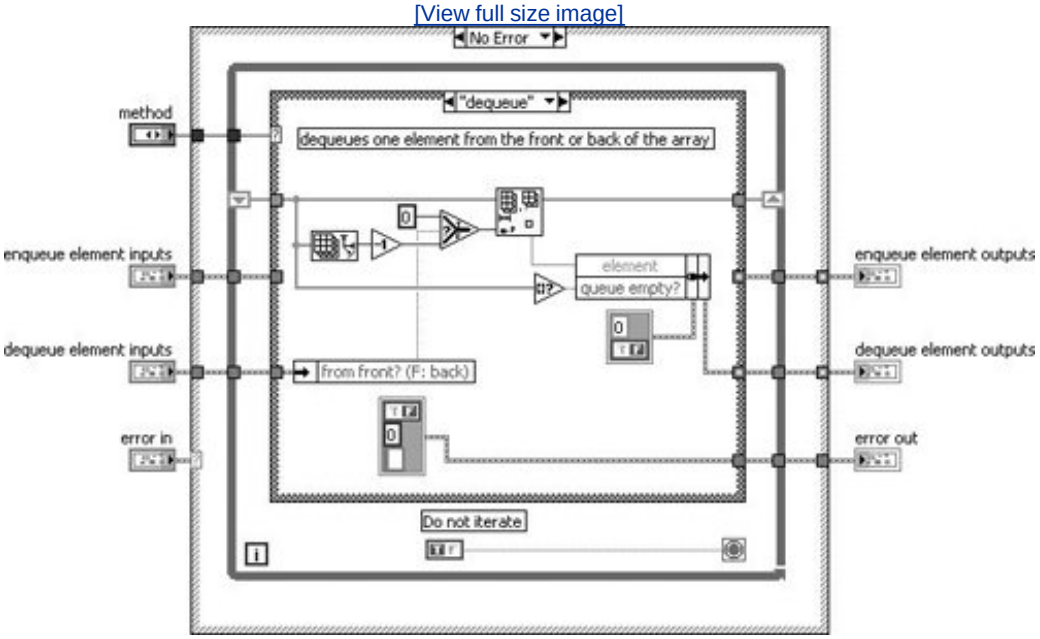


Figure D.6. Functional Queue.vi's "dequeue" method case

[View full size image]



Functional Global "Public Method" VIs

The functional global "core" VI that we have just created, Functional Queue.vi, contains all of the object's methods and data. However, it is quite cumbersome and confusing to use as a subVI. For example, the inputs and outputs that are used depend on the value passed to the `method` enum.

Additionally, it would be very difficult to debug applications that call Functional Queue.vi directly because we cannot easily find all instances that are called with a specific `method` value. So, in order to make using Function Queue.vi practical, we create a "wrapper" VI for each method. These will be our object's "public method VIs," the VIs that users of our functional global object will call as subVIs. Figure D.7 through Figure D.9 show the three public method VIs of our object: Init Queue.vi, Enqueue.vi, and Dequeue.vi. As you can see, each VI has individual inputs and outputs and does not use the input and output clusters.

Figure D.7. Init Queue.vi public method VI

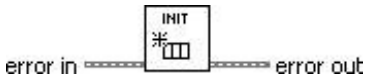


Figure D.8. Enqueue.vi public method VI

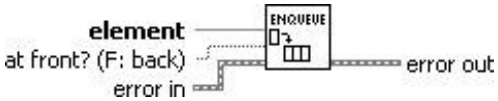
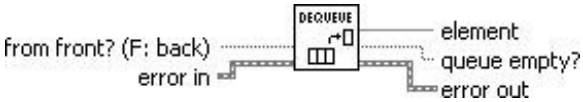


Figure D.9. Dequeue.vi public method VI



Now let's take a look at the block diagrams of our public method VIs. Figure D.10 through Figure D.12 show the block diagrams of Init Queue.vi, Enqueue.vi, and Dequeue.vi. Note that each of these VIs sets the `method` enum and handles the bundling and unbundling of the input and output clusters (respectively).

Figure D.10. Init Queue.vi block diagram

[View full size image]

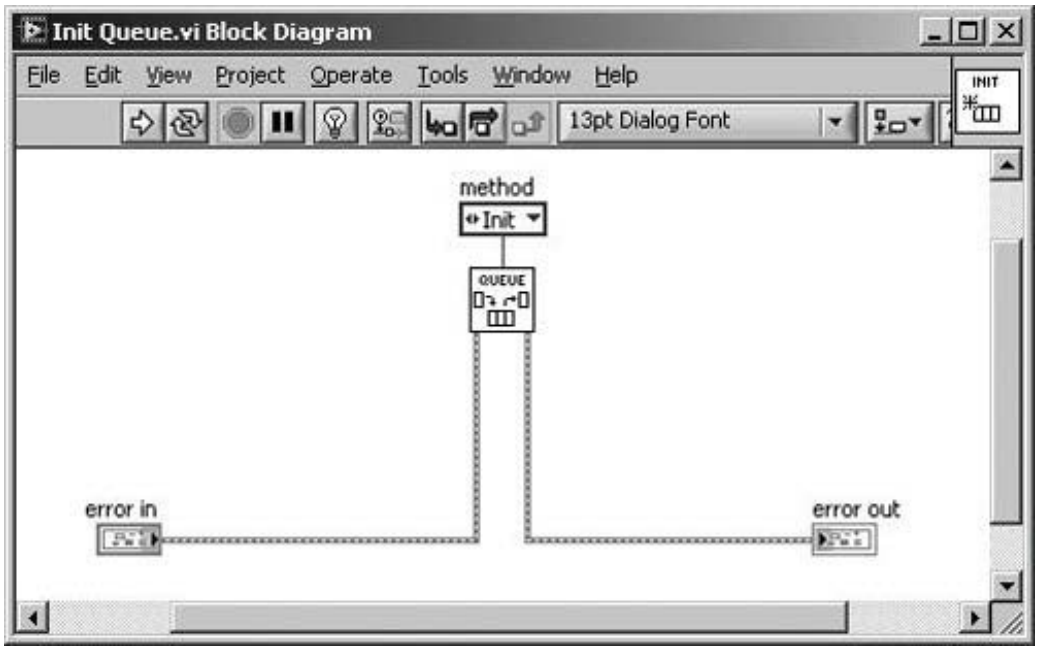


Figure D.11. Enqueue.vi block diagram

[View full size image]

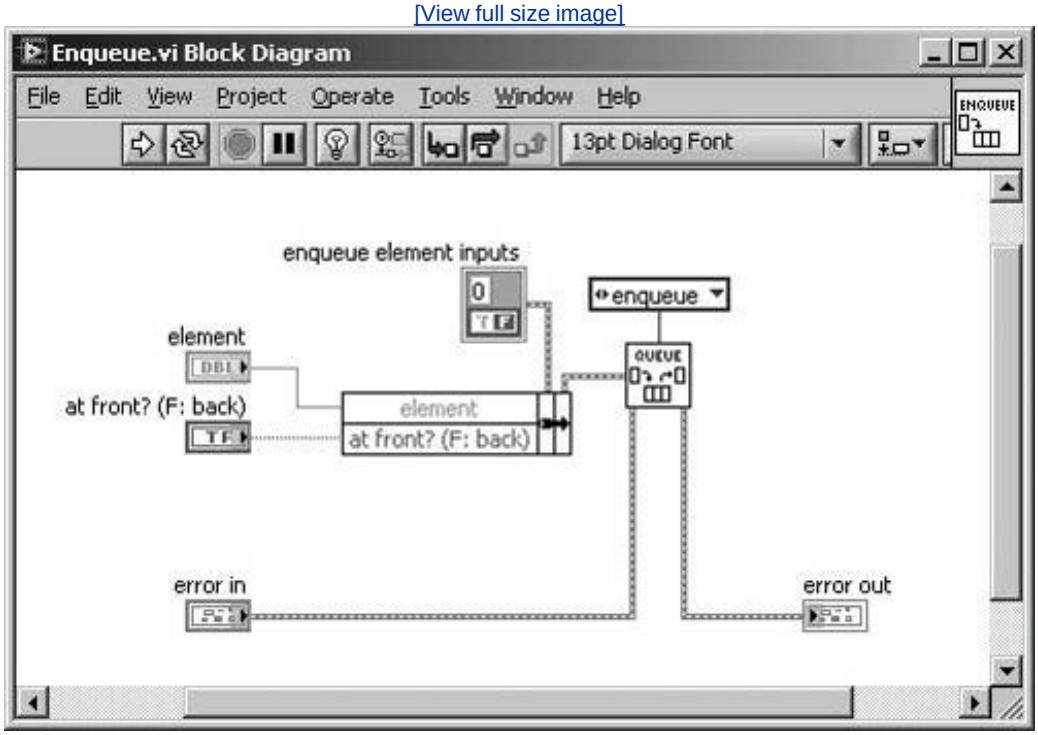
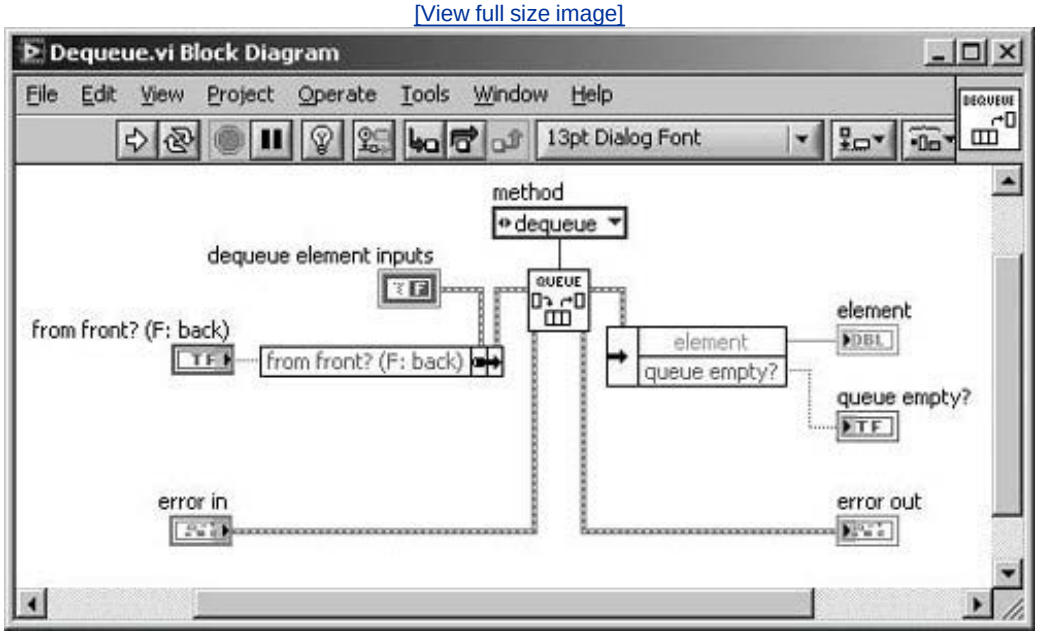


Figure D.12. Dequeue.vi block diagram

[View full size image]

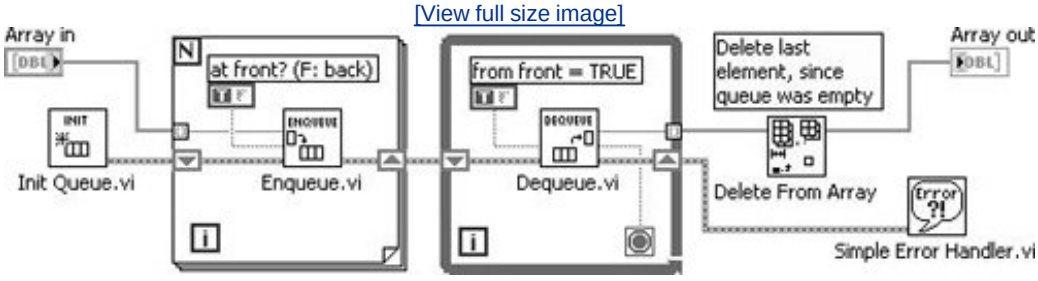


Programming Applications Using the Public Methods

Now let's look at an example of how we might use our object. Figure D.13 shows the block diagram of a VI named Reverse Array.vi that uses the queue object to reverse the order of an array. First, each of the array elements is enqueued onto the front of the queue in a For Loop. Then, each of the array elements is dequeued from the front of the queue in a While Loop, until the queue is empty. In this example, we are treating the queue as a FIFO, which will cause the output array to be the reverse of the input array.

Figure D.13. Reverse Array.vi block diagram

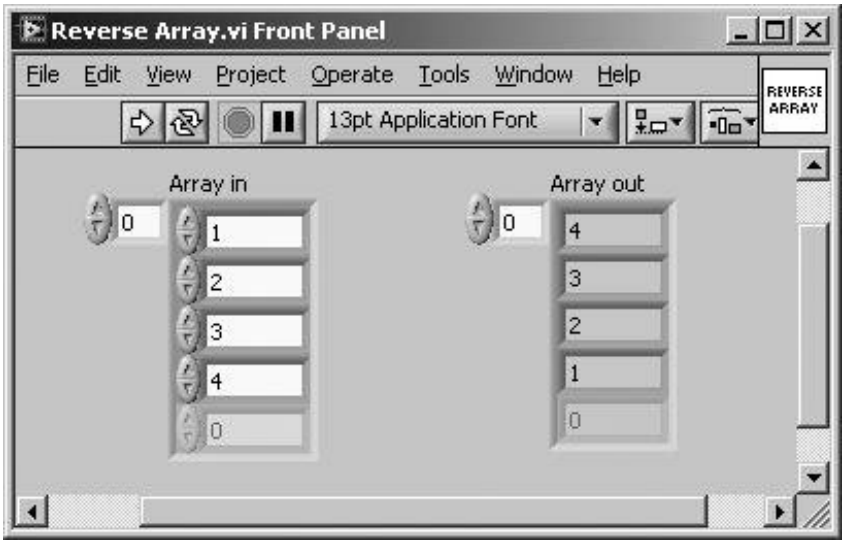
[View full size image]



Looking at the front panel of Reverse Array.vi, shown in Figure D.14, populated with an input array and showing the output array after the VI is run, you can see that, in fact, the output array is the

reversed input array.

Figure D.14. Reverse Array.vi front panel



Functional Globals: Final Thoughts

In summary, it is important to remember the following points about functional globals:

- The functional global core contains the object data and methods.
    - The `method` enum contains one element for each method.
    - Each method has an input and output cluster.
    - Each method unbundles its own input cluster and bundles its own output cluster.
    - The method enum, input clusters, and output clusters should all be type definitions.
    - Make sure to initialize your object data inside the "Init" case.
    - The functional global core should not be reentrant. Otherwise, the public method VIs would each call a unique instance of the core.
- The functional global public methods are used to program your application. They are the programmer's "interface" to the object.
    - The functional global public methods are the only VIs that call the functional global core as a subVI.

- There is usually one public method for each method defined in the `method` enum and implemented inside the core's case structure.
- The public methods set the `method` enum when calling the functional global core.
- The public methods bundle the appropriate method's input cluster, from control elements on its front panel, and pass it into the core.
- The public methods unbundle the appropriate method's output cluster and pass elements to output indicators on its front panel.

Note that functional globals implement the *Class*, *Object*, and *Encapsulation* concepts mentioned at the beginning of this appendix. Functional globals can be safely called an *object-based* programming technique, because they do not implement all the object-oriented concepts. The encapsulation provides modular application components that hide the details of how each method does its job and how the object stores its data. This allows you to modify and improve the functional global without affecting code that uses it (unless, of course, you change inputs, outputs, or behavior of the public methods).

GOOP

GOOP is almost an *anachronym*people use it so casually, they often forget that it is an acronym for Graphical Object-Oriented Programming. GOOP generally refers to a pattern of object-oriented programming that uses a "semaphored" object data store that allows multiple object instances "by reference." There are several implementations of GOOP, all of which use a *wizard* to generate and help you edit classes from a template. Each of these implementations of GOOP provide support for the *Class*, *Object*, and *Encapsulation* object-oriented concepts mentioned at the beginning of this appendix.

So what do "semaphored object data store" and "multiple object instances by reference" mean? The next sections will attempt to shed some light on these concepts.

Semaphored Data Store

In Chapter 13, you learned that semaphores are used to lock and unlock access to a shared resource. An object data store is certainly a shared resourceit is shared by all the class's methods, because each method needs to access the object's data. In the case of a functional global, which we learned about in the last section, each method is a separate case of a case structure within the functional global core. And, because only one case of the Case Structure can execute at a given time, there is no chance that two methods will attempt to access the object data at the same time. Recall, the functional global core is *not reentrant*, so if two different public methods are running in parallel, only one of them can call the functional global corethe other public method VI will have to wait until the call in the other public method VI has completed. However, if we had a situation where two methods that operate on the object data might be able to run at the same time, we would need a mechanism to lock, or semaphore, the object data.

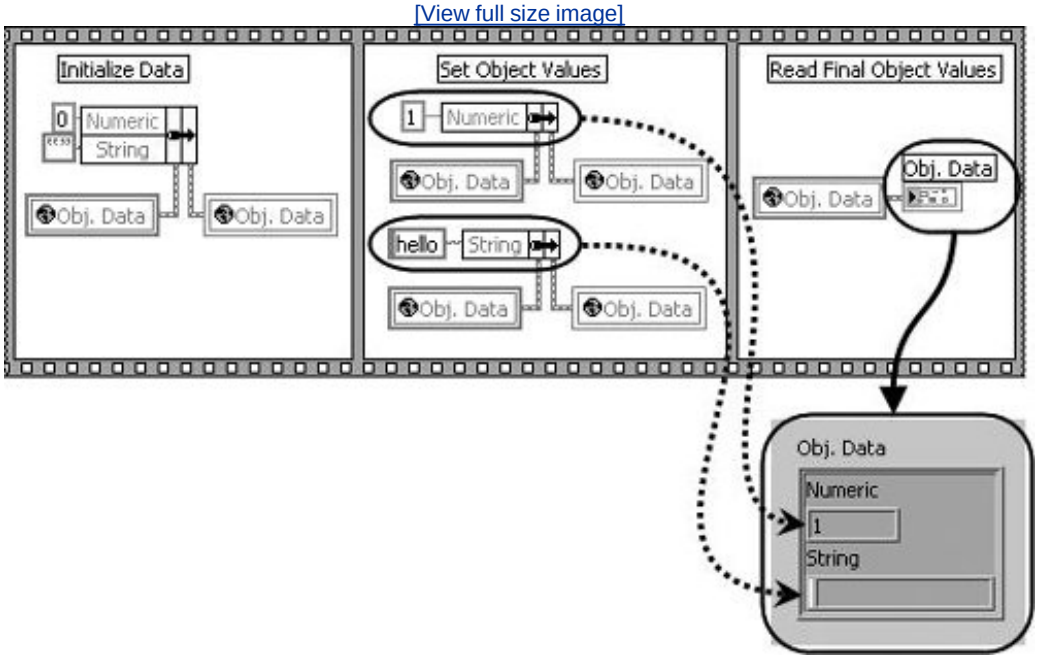
Recall from the "Semaphores: Locking and Unlocking Shared Resources" section of Chapter 13, that we stated something to the effect of, "*Semaphores having data is the fundamental concept of GOOP."* Queues and notifiers have data, but semaphores do not. However, imagine that semaphores

have data, and voilàyou have GOOP.

Let's now look at an example of why semaphores are important when accessing shared data. Notice in Figure D.15 that although we have set the value of Numeric and String, only Numeric actually remains in the Obj. Data once the code completes execution. What has happened is a classic *race condition*, where two parallel operations read data, operate on it, and then write back the modified data. In our example, the operation on top a has overwritten the result of the operation on the bottom. In order for us to remedy this problem, we must ensure that the upper and lower operations occur sequentially and not in parallel.

Figure D.15. Unsemaphored GOOP Global.vi block diagram and front panel (after running)

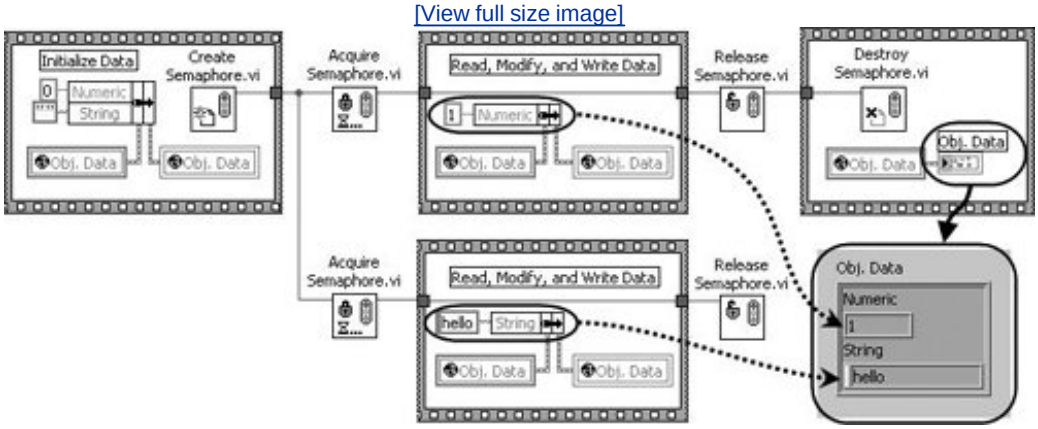
[View full size image]



Of course, we could remedy this problem by putting the two data modification steps into separate frames of the sequence structure, but what if these operations are happening in completely different VIs in completely different parts of your application? It is easy to see that this could get tricky in a hurry. But, by using semaphores to force sequential access to our shared data, we can ensure that only one data modification operation occurs at a given time. As you can see in Figure D.16, by using a semaphore to enforce sequential access to our shared data structure, *the upper and lower data modification operations are no longer executed in parallel*. Both data modification operations are successful, as we can see by the presence of the updated Number and String values in the final value of `Obj. Data`.

Figure D.16. Semaphored GOOP Global.vi block diagram and front panel (after running)

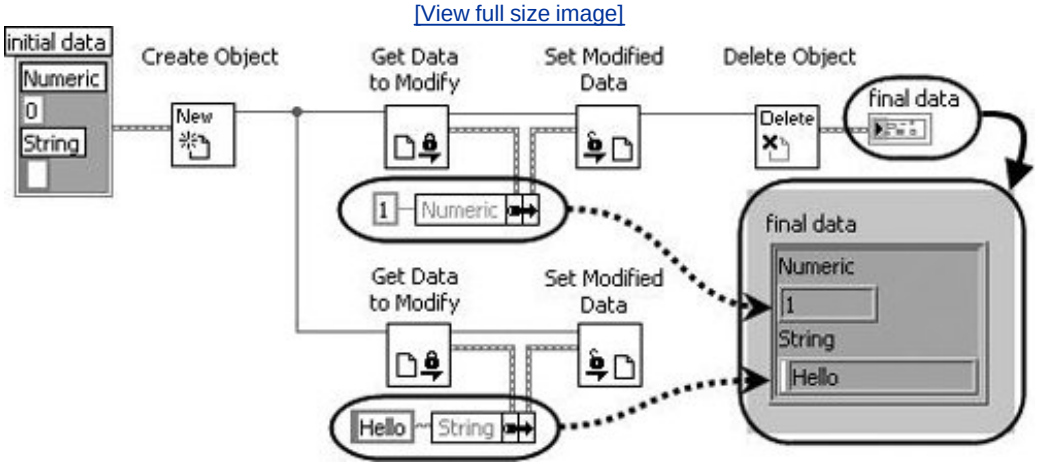
[View full size image]



In the example we just looked at, we used a global variable to store our object data. And, the upper and lower data modification operations were our methods. The semaphore was used to ensure that only one method could operate on the data at a given time. Now, let's take this concept another step forward and more closely couple the semaphore to the object data (remember, semapore + data = GOOP). Figure D.17 shows how the internal components of a GOOP class (created using the NI GOOP Wizard) are used to control access to the shared object data. Note that the data flows directly out of the Get Data to Modify VI and directly into the Set Modified Data VI. These two VIs are analogous (and similar in behavior) to the Acquire Semaphore and Release Semaphore VIs, but with the addition of object data (remember, semapore + data = GOOP).

Figure D.17. GOOP Class Example.vi block diagram

[View full size image]

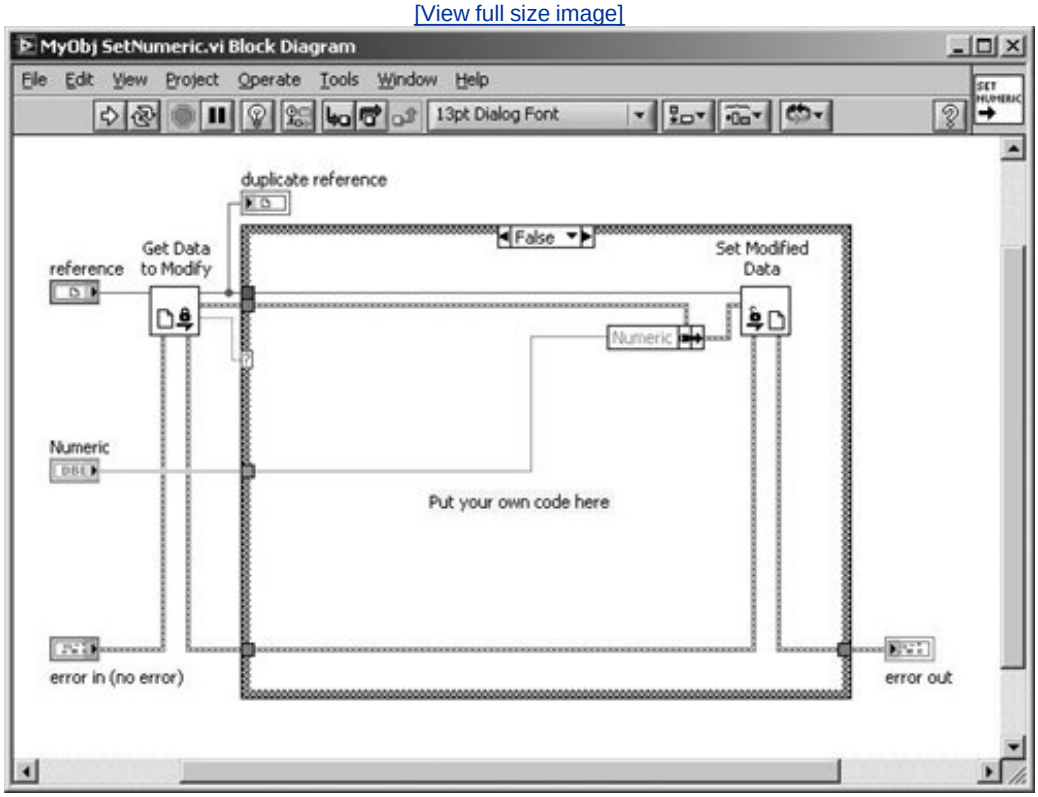


The previous example (shown in Figure D.17) contains internal support VIs used by GOOP classes. These internal VIs are not intended to be called by developers programming applications that use the class; rather, they are intended to be called as subVIs inside of *class method VIs*.

Figure D.18 shows an example of a class method VI named MyObj SetNumeric.vi. This VI is a member of the "MyObj" class and is the SetNumeric method. Its purpose is to set the value of the `Numeric` element of the object data. It is inside of the method VIs where we perform work based on method arguments and object data. While inside of a method that modifies the data, we have exclusive access to the object datano other instance of Get Data to Modify will be able to access the data until the data is written back to the data store using Set Modified Data (which, again, is analogous to the behavior of Acquire Semaphore and Release Semaphore).

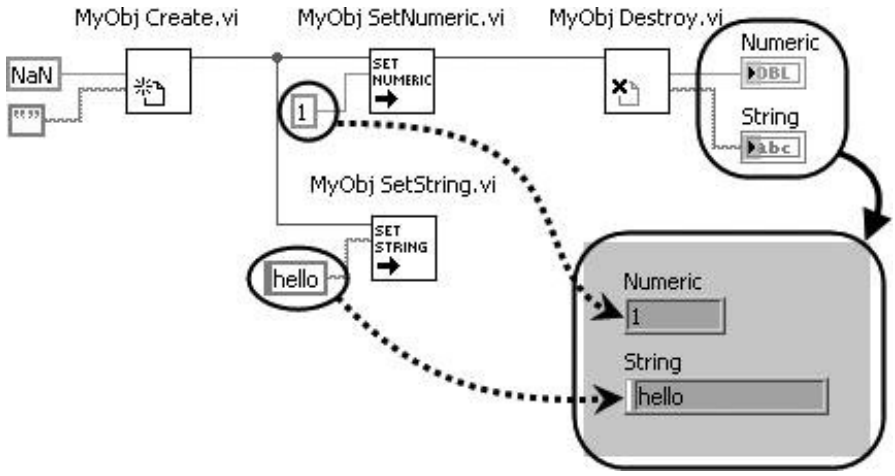
Figure D.18. MyObj SetNumeric.vi block diagram

[View full size image]



Now, using class method VIs, as shown in Figure D.19, our example becomes much simpler. Note that MyObj SetString.vi works in a similar fashion as MyObj SetNumeric.vi. And, MyObj Create.vi and MyObj Destroy.vi are simple VIs that call the Create Object and Delete Object VIs shown in Figure D.17these are referred to as the *object constructor method* and *object destructor method* (respectively), in object-oriented terminology.

Figure D.19. GOOP Class Method Example.vi block diagram



It should also be noted that it is possible to create methods that do not modify the data. If this is the case, they call the utility VI named Get Data, which (unlike Get Data to Modify) *does not acquire the object semaphore*. This should only be used inside of methods that do not modify the data. The benefit of using Get Data is that (unlike Get Data to Modify) *it does not wait for the object data semaphore to be released*it returns immediately with the latest object data. Methods that call Get Data (and therefore do not modify the object data) are referred to as *reading* or *read-only* methods. Methods that call Get Data to Modify (and do modify the object data) are referred to as *modifying* or *read-write* methods. Note that *only one modifying/read-write method will be able to execute at a time*, due to the locking/unlocking provided by the object semaphore.

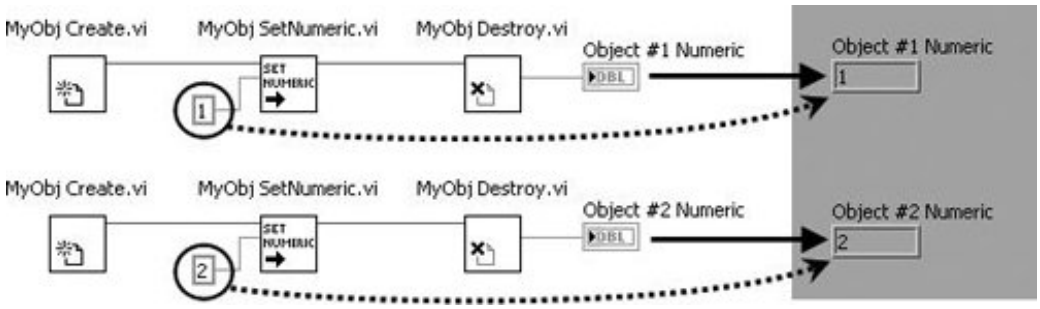
Multiple Object Instances by Reference

Up to now, we have seen only single instances of classesone object per class. However, using GOOP, we can create multiple objects (instances of our classes) simply by calling the class's Create (constructor) VI multiple times. Each time we call the Create VI, a new, unique object reference will be generated. This is similar to how each time one calls the Create Semaphore VI, a new, unique semaphore reference is generated.

The block diagram shown in Figure D.20 shows a simple example of how you can create multiple object instances, by simply calling the constructor method multiple times. Yes, that's all there is to it. Each reference refers to a unique object with its own separate data.

Figure D.20. GOOP Multiple Instances.vi block diagram

[View full size image]



Obtaining the GOOP Wizard

By this point, you're probably excited to try GOOP, and we bet you're wondering how you can make your own GOOP classes. Well, it's pretty easysimply download the GOOP Wizard from ni.com. A simple search will help you find it. Additionally, there are many other third-party GOOP wizards (for example, OpenGOOP from OpenG.org, as discussed in Appendix C, "Open Source Tools for LabVIEW: OpenG")each of them uses a slightly different approach to how data is stored in memory. For GOOP beginners, we recommend using the NI GOOP Wizard, because it's easy to use and supported by NI.

GOOP Final Thoughts

GOOP is one evolutionary step forward from functional globals, adding support for multiple object instances by reference and also eliminating the need for a single "core" VI that contains every method. However, this comes with a price. First, GOOP is less efficient (it's hard to beat the read/write performance you gain by storing data in a shift register, as is done in functional globals). Second, GOOP classes have many "internal" support VIs that are used for managing access to the object data. It is not uncommon for each class to require 0.5 MB to 2 MB of support VIs, depending on which version of GOOP you are using. And, all of these support VIs can create a bit of a housekeeping problem considering each VI is a part of your project. Another issue is renaming classes. The NI GOOP Wizard, for example, does not support renaming classes. However, there are several tools that people have written to support these development tasks. Another excellent example is the GOOP Wizard from Endevo (www.endevo.se)the people at Endevo are the inventors of GOOP, by the way. They sell some very powerful GOOP tools with many great features. And, they have a more recent version of GOOP that supports inheritance, which we will learn about next.

GOOP Inheritance

We mentioned in the last section that GOOP provides support for the *Class*, *Object*, and *Encapsulation* object-oriented concepts. However, there are also implementations of GOOP frameworks that also provide support for *Inheritance*, *Abstraction*, and *Polymorphism*. (Note that object-oriented polymorphism is not at all the same thing as LabVIEW's polymorphic VIs, whose use of the term is a bit of a misnomer.) With these additions, GOOP Inheritance becomes a very complete object-oriented framework. The only drawbacks of these frameworks are the following:

- Each class has even more internal support VIs than "regular" GOOP classes.

- Data access is generally slower, because there are a few more hoops to jump through in order to access inherited data.

- Editing operations due to increased inherited dependencies can be more complex and really do require additional editing tools in order to edit GOOP Inheritance classes.

An in-depth discussion of GOOP Inheritance is beyond the scope of this appendix. However, you are encouraged to learn more about it. Endevo (www.endevo.se) has a very good downloadable white paper describing the fundamentals of its GOOP Inheritance product. Additionally SciWare (www.sciware.com.au) sells a GOOP Developer product that supports Inheritance and also has good documentation on its use.

PREV NEXT

Built-in LabVIEW Object-Oriented Programming Features

For quite a long time, LabVIEW developers have needed and wanted built-in (native) support for object-oriented programming. The benefits of having OOP built into LabVIEW are the lack of need for internal support VIs, built-in support for editing tasks, editor/compiler support for enforcing encapsulation rules, better efficiency of execution, and so on. With LabVIEW 8.0 came the first built-in features that addressed object-oriented programming, and subsequent LabVIEW releases are extremely likely to continue down this path toward native support for OOP. With LabVIEW 8.0 came two new object-oriented(ish) features: *project libraries* and *XControls*.

Project Libraries

We discussed project libraries, very briefly, in Chapter 16, "Connectivity in LabVIEW," when we discussed *shared variables* (because shared variables must be members of project libraries). But, project libraries are good for a lot more than just storing shared variables. You can specify library member VIs as being *public* or *private*. Private VIs may only be called as subVIs by members of the project library, whereas public VIs may be called by any VIs in your application (public VIs are your project library's *interface*). This is a huge step forward toward native support for object-oriented programming in LabVIEW. It is the first built-in feature of the LabVIEW editor that implements one of the six object-oriented concepts (*Encapsulation*). There are a variety of indications, made publicly by National Instruments, that built-in object-oriented features are just around the corner and will likely evolve from the project pibrary.

XControls

XControls are a feature that allows you to create controls and indicators that have edit-time behavior that is defined by LabVIEW code that you write. And, just like traditional controls and indicators, you can define properties and methods for your XControls. You create instances of XControls by placing them on the front panel of a VI. If you want multiple instances, place multiple instances of your XControl. Although this is not intended to be a built-in object-oriented programming construct, the object-orientedness of the design is worth highlighting.

The Future of LabVIEW Object-Oriented Programming

Although there are still five out of six object-oriented concepts that are not currently supported natively in LabVIEW (project libraries provide *Encapsulation*), we are *very* confident that the future of object-oriented programming in LabVIEW looks very bright. And that's all we're going to say. ;-)

E. Resources for LabVIEW

You have many options for getting help and finding additional information about LabVIEW and virtual instrumentation. The following is a list of web sites, organizations, publications, and other resources for LabVIEW users.

LabVIEW Documentation and Online Help

Sometimes it's easy to overlook the obvious places to get help. The LabVIEW online and printed manuals provide excellent tutorials to get you up to speed with LabVIEW. They also contain plenty of reference information. You can find the answers to almost all of your questions in these manuals. In addition, LabVIEW has extensive online help to assist you as you build your application.

The Guys Who Wrote the Book

When all else fails, you can always hire "one of the guys who wrote the book" to help. Both Jim Kring and Jeffrey Travis offer expert LabVIEW consultation, training, and support through their respective consulting firms.

Here is their contact information:

Jeffrey Travis Studios LLC
5409 Aurora Drive
Austin, Texas 78756
USA
Voice: 512-371-3614
Fax: 512-697-0040
Web: http://jeffreytravis.com
Email: consulting@jeffreytravis.com

James Kring, Inc.
1798 Great Hwy, #3
San Francisco, CA 94122
USA
Voice: 888-891-7821 (toll free)
Fax: 415-366-3299
Web: http://jameskring.com
Email: info@jameskring.com

National Instruments

National Instruments is the company that creates and sells LabVIEW. They offer extensive technical support by phone, email, and their web site. You can contact them at

National Instruments Corporation
11500 N Mopac Expwy
Austin, Texas 78759-3504

USA
Phone: 800-531-5066
Fax: 512-683-8411
Web: http://ni.com
Technical resource site: http://zone.ni.com

LabVIEW Advanced Virtual Architects (LAVA)

LabVIEW Advanced Virtual Architects, or *LAVA*, is an independently run web site that contains numerous resources for intermediate to advanced LabVIEW users. Browse through the lively forum discussions to learn cutting-edge programming techniques guaranteed to shave months off your development time. The forum contributors are among some of the most advanced LabVIEW programmers. If you've gone through the basics and want to take your programming to the next level, a visit to this site is a must. For more information, visit the LAVA web site at http://www.lavag.org.

Info-LabVIEW Mailing List

Info-LabVIEW is a user-sponsored Internet mailing list that you can use to communicate with other LabVIEW users. It is probably one of the best unbiased resources you can find on LabVIEW. You can post messages containing questions, answers, and discussions about LabVIEW on this mailing list. These messages will be sent to LabVIEW users worldwide.

To subscribe, send an email to info-labview-on@labview.nhmfl.gov for individual messages, or info-labview-digest@labview.nhmfl.gov for daily digests. All messages posted to Info-LabVIEW will then be forwarded to your e-mail address. You can cancel your subscription by sending a message to the preceding address requesting that your email address be removed from the list.

You can get more information about Info-LabVIEW at http://www.info-labview.org.

You can also search the Info-LabVIEW archives at http://www.searchview.net.

OpenG.org

OpenG is an organized community committed to the development and use of open source LabVIEW tools, applications, frameworks, and documentation. They provide free-as-in-speech toolkits for LabVIEW and provide forums for discussions related to OpenG tools for LabVIEW. See Appendix C, "Open Source Tools for LabVIEW: OpenG," for more information about OpenG and open software LabVIEW tools.

The OpenG.org web site is at http://openg.org.

Other Books

There are a number of good books that explore specific LabVIEW topics or application areas in much further detail than *LabVIEW for Everyone;* the following are some that are recommended:

LabVIEW Advanced Programming Techniques by Rick Bitter, Taqi Mohiuddin, and Matthew Nawrocki. CRC Press, 2000. ISBN: 0849320496.

LabVIEW Graphical Programming by Gary Johnson and Richard Jennings. McGraw-Hill, 2006. ISBN: 0071451463.

LabVIEW GUI: Essential Techniques by David J Ritter. McGraw-Hill, 2002. ISBN: 0071364935.

The LabVIEW Style Book by Peter Blume. Prentice Hall, 2006. ISBN: 0131458353.

A Software Engineering Approach to LabVIEW by Jon Conway and Steve Watts. Prentice Hall, 2003. ISBN: 0130093653.

PREV NEXT

F. LabVIEW Certification Exams

Obtaining certification in LabVIEW development is an excellent way to validate your LabVIEW skills. Many employers provide incentives for employees to obtain certification, and certifications certainly look impressive on your résumé. But also, the process of studying for a certification exam will often help you identify areas where you may be lacking some experience. You can then improve on those areas and ensure that you have a well-rounded understanding of LabVIEW. Additionally, once you are certified, you will be recognized on ni.com, and will be able to use the LabVIEW certification logos (e.g., the Certified LabVIEW Associate Developer logo shown in Figure F.1) for your business and résumé.

Figure F.1. A LabVIEW certification logo

National Instruments offers the following levels of LabVIEW certifications (shown in the order that they must be achieved):

1. Certified LabVIEW Associate Developer (CLAD) Validates the foundational knowledge and skills required to develop and maintain LabVIEW applications.
2. Certified LabVIEW Developer (CLD) Validates the knowledge and skills to design, develop, and deploy a scalable, readable, and maintainable LabVIEW application using advanced software principles, architectures, techniques, and the LabVIEW Development Guidelines. *Successful completion of the CLAD is required before you can take the CLD exam.*
3. Certified LabVIEW Architect (CLA) Validates the knowledge and skills necessary, given a set of requirements for a large application, to develop, lead, and direct a team of LabVIEW developers in the creation of an efficient, cost-effective solution. *Successful completion of the CLD is required before you can take the CLA exam.*

You can find information on NI's training and certification program, as well as sign up to take the

exams, at http://ni.com/training/.

PREV NEXT

Certified LabVIEW Associate Developer (CLAD) Exam

The CLAD exam is the first step in the certification process. The test is proctored by *Pearson VUE*, at testing centers around the world. This is a short, multiple-choice test that is taken at a computerized testing kiosk.

Preparing for the CLAD

You can prepare for the CLAD exam by doing the following:

- Make sure that you have read and completed the activities in the *Fundamentals* section of *LabVIEW for Everyone* (Chapters 1 through 9) and have paid close attention to areas with CLD topic callouts (with a CLD icon in the margin like the one shown on the left).

- Review the CLAD Exam Topics.pdf document located in the `CERTIFICATION` folder of the CD-ROM. Make sure that you understand these topics. Find these topics in *LabVIEW for Everyone* and pay close attention to areas with CLD topic callouts.
- Take the CLAD LabVIEW Fundamentals Exam (see the next section).
- Take the CLAD Sample Exam (CLAD Sample Exam.pdf is located in the `CERTIFICATION` folder of the CD-ROM).
- See the CLAD Exam Preparation Resources.pdf document located in the `CERTIFICATION` folder of the CD-ROM for additional preparation resources.
- Take the LabVIEW Basics I and Basics II training courses offered by National Instruments and NI Certified Training Facilities.

Test Yourself: The LabVIEW Fundamentals Exam

The best way to begin preparation for the CLAD exam is to first take the free, online preparation exam, the LabVIEW Fundamentals Exam. This preparatory exam may be found at http://www.ni.com/training/labview_exam/ or by following the links on the http://www.ni.com/training/ page. This exam consists of 40 multiple-choice questions with a 45-minute time limit. It encompasses essential LabVIEW concepts, such as the following:

- Arrays and clusters

- Charts, graphs, and loops
- Data acquisition and analog input
- General programming structures
- Strings and error handling
- SubVIs and printing

The results of this exam will help you identify whether you are ready to take the CLAD exam, or whether you should review certain subject areas.

Note that the LabVIEW Fundamentals Exam page on ni.com has links to several good resources that will help you prepare for the exam.

PREV NEXT

Certified LabVIEW Developer (CLD) Exam

After you have successfully achieved CLAD certification, you are eligible to take the CLD exam, the second level of LabVIEW certification. The test is a practical exam, having questions that require you to create LabVIEW VIs that demonstrate your proficiency, knowledge, and skills to design, develop, and deploy scalable, readable, and maintainable LabVIEW applications. You are expected to know and use advanced software principles, architectures, techniques, and the LabVIEW Development Guidelines.

You can prepare for the CLD exam by doing the following:

- Pass the CLAD exam (required).
- Take the LabVIEW Fundamentals Exam (see the previous section) again to make sure you remember the CLAD material.
- Make sure that you have read and completed the activities in the *Advanced* section of *LabVIEW for Everyone* (Chapters 10 through 17) and have paid close attention to areas with CLD topic callouts (with a CLD icon in the margin like the one shown on the left).

- Take the CLD Sample Exam (CLD Sample Exam.pdf is located in the `CERTIFICATION` folder of the CD-ROM).
- Take the LabVIEW Intermediate I and Intermediate II training courses offered by National Instruments and NI Certified Training Facilities.

You absolutely must practice for the CLD exam by taking the practice teststhere is no better way to ensure that you are ready. You must be able to complete these practical exercises within the required time frame (four hours). One thing that will help you enormously is knowing how to create and use a state machine (SM) or queued message handler (QMH), as discussed in Chapter 13, "Advanced LabVIEW Structures and Functions."

Practice creating and using the SM and QMH structures. Memorize their components and practice creating these structures from scratch, purely from memory. (You won't be able to

bring your copy of LabVIEW for Everyone or any example VIs with you to the exam, so you had better know how to create these from memory.) Time yourself to see how quickly you can create a SM or QMH "template"it shouldn't take you more than five or ten minutes. When you sit down to take the CLD exam, the first thing you should do (after reading the exam question) is create a SM or QSM template for use during the exam. Save a copy of it before starting to work on the exam, just in case you decide to start over. Back up your work regularly, in case you have computer problems.

PREV NEXT

Certified LabVIEW Architect (CLA) Exam

Passing the CLA exam is like getting your "black belt" in LabVIEW. And, just like getting your black belt in karate, it take years of hard work and learning things that just aren't taught in books (well . . . not in just one book, alone). It also requires using LabVIEW often and for a long period of time over several projects and on teams with multiple developers. The goal of the CLA is to validate that, given a set of requirements for a large application, one is able to develop, lead, and direct a team of LabVIEW developers in the creation of an efficient, cost-effective solution.

You can prepare for the CLA exam by doing the following:

- Pass the CLD exam (required).
- Review the CLA Topics (CLA Topics.pdf is located in the `CERTIFICATION` folder of the CD-ROM).
- Dedicate yourself to the pursuit of LabVIEW knowledge.
- Use LabVIEW passionately for many years on large projects and with multiple developers using source code control tools and semi-formal software project documentation.
- Ask and answer questions on the various online discussion forums and lists. (See Appendix E, "Resources for LabVIEW," for more info.)
- Take the LabVIEW Advanced Application Development course offered by National Instruments and NI Certified Training Facilities.
- Practice writing LabVIEW code with your eyes closed. (OK, we made this one up, but wouldn't that be cool?)

Glossary

Symbols

∞

Infinity.

p

Pi.

Delta; difference. x denotes the value by which x changes from one index to the next.

.NET

A framework from Microsoft that allows Windows applications to communicate with one another and to embed user interface components from one application into another. See also *Active X*.

\ codes display

A display mode for string controls, indicators, and constants that displays non-printable characters as a slash ("\") followed by the non-printable character's hexadecimal value (or special character).

A

Absolute Path

File or directory path that describes the location relative to the top level of the file system.

Active Window

Window that is currently set to accept user input, usually the frontmost window. 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.

ActiveX

A framework from Microsoft that allows Windows applications to communicate with one another and to embed user interface components from one application into another. See also *.NET*.

A/D

Analog-to-digital conversion. Refers to the operation electronic circuitry does to take a real-world analog signal and convert it to a digital form (as a series of bits) that the computer can understand.

ADC

See *A/D*.

alignment grid

Grid lines that can be shown on the front panel and block diagram windows, which allow easy alignment of objects, at edit-time.

ANSI

American National Standards Institute.

apple events

A framework from Apple that allows Mac OS X (and earlier versions of Mac OS) applications to communicate with each other and with the operating system.

application builder

A component of LabVIEW that allows building stand-alone, executable applications. See also *Build Specification*.

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.

artificial data dependency

Condition in a dataflow programming language in which the arrival of data, rather than its value, triggers execution of a node.

ASCII

American Standard Code for Information Interchange. Refers to a seven-bit encoding scheme for alphanumeric characters.

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.

automatic error handling

A LabVIEW feature that captures errors and displays a dialog, if an error occurs inside a subVI or function whose error cluster output is unwired on the block diagram.

automatic tool selection

If automatic tool selection is enabled and you move the cursor over objects on the front panel or block diagram, LabVIEW automatically selects the corresponding tool from the Tools palette.

automatic wire routing

If automatic wire routing is enabled, LabVIEW automatically tries to determine a route for a wire as you connect it to a terminal or node.

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.

autotool

See *Automatic Tool Selection*.

auto wiring

If auto wiring is enabled, LabVIEW will automatically wire block diagram objects that are placed in close proximity to each other, if they have inputs and outputs of that same type.

B

binary file

A file that contains flattened data. A binary file is more efficient than a text file in disk space and in speed. See also: *Flattened Data*, *Text File*.

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

A datatype that has two states: TRUE or FALSE. In LabVIEW, Booleans are represented by the color green.

boolean controls

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. You set a breakpoint by clicking on a VI, node, or wire with the Breakpoint tool from the Tools palette.

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.

broken wires

Wires that cannot work properly because they are not connected to anything, or are connecting mismatched data types, or are not connected between one and only one data source and at least one data sink. They appear as dashed lines, instead of solid and colored (according to a LabVIEW data type color).

buffer

Memory for storing data. For example, a buffer is used for transferring data from a DAQ device to one's LabVIEW application.

build specification

A set of rules for transforming project source files into a built software product, such as an executable, installer, ZIP file, shared library, or source distribution. A LabVIEW project can have multiple build specifications.

bundle node

Function that creates clusters from various types of elements. See also *Unbundle Node*.

byte stream file

File that stores data as a sequence of ASCII characters or bytes.

PREV NEXT

C

call library function node

A LabVIEW node that calls a DLL or shared library function directly. See also: *Shared Library*, *DLL*.

caption label

Similar to a Name Label, but may be changed programmatically at run-time. See also *Name Label*.

case

One subdiagram of a Case Structure, Event Structure, Conditional Disable Structure, or Diagram Disable 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.

channel

Pin or wire lead to which an analog signal is read from or applied.

chart

See *Scope Mode*, *Strip Mode*, and *Sweep Mode*.

CIN

See *Code Interface Node*.

client-server

A relationship between applications where an application (a client) uses services provided by another application (a server), usually over a network, but sometimes on the same computer.

clone

An instance of a Reentrant VI. See also: *Reentrant Execution*, *Reentrant VI*.

cloning

To make a copy of a control or some other LabVIEW object by clicking the mouse button while pressing the <ctrl> (Windows); <option> (Mac OS X); <meta> (Linux) key and dragging the copy to its new location.

(Linux) 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. A cluster's elements are always either all controls or all indicators.

cluster shell

Front panel object that contains the elements of a cluster.

Code Interface Node (CIN)

Special block diagram node through which you can link conventional, text-based code (such as C++) 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 copy tool

Copies colors for pasting with the Color tool.

color tool

Tool you use to set foreground and background colors.

Comma Separated Values (CSV) File

A *Spreadsheet File* in plain text format that uses a comma character as the delimiter to separate fields. See also *Spreadsheet File*. Commonly, CSV files are saved with a .csv file extension.

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.

compiled help

A .chm or .hlp file that contains help documentation (Windows only).

conditional disable structure

A structure with one or more subdiagrams, or cases, exactly one of which LabVIEW uses for the duration of execution, depending on the configuration. Commonly used for switching between multiple platform-specific function calls.

conditional terminal

The terminal of a While Loop containing a Boolean value that determines whether the VI performs another iteration.

config file

See *INI File*.

configuration file

See *INI* File.

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 window that displays the VI terminal pattern. It underlies the icon pane.

constant

See *Universal Constant* and *User-Defined Constant*.

context help window

A floating window that displays basic information about LabVIEW objects when you move the cursor over each object.

continuous acquisition

A data acquisition task that runs continuously, filling a buffer that is continuously read by the application (LabVIEW) software.

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 description

See *Description Property*.

control editor

A window that allows customizing controls and indicators. Used for editing and creating *Custom Controls*, *Type Definitions*, and *Strict Type Definitions*.

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.

control reference

A *VI Server Reference* to a front panel control or indicator. Can be used with a *Property Node* or *Invoke Node*.

count terminal

The terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.

counter

Hardware that can measure timing information related to a digital input signal and can generate digital output signals with specified timing information.

CPU

Central processing unit.

Current VI

VI for which the front panel, block diagram, or Icon Editor is the active window.

cursor

The icon which represents the mouse location on a computer screen.

custom control

A LabVIEW control file (.ctl file) which is a definition of a control or indicator. See also *Control Editor*, *Type Definition*, and *Strict Type Definition*.

custom help

Help documentation created for VIs, which is associated with the VI, so that it can be accessed from the *Context Help* window.

Custom PICT Controls

Controls and indicators for which parts can be replaced by graphics and indicators you supply.

custom probe

A debugging tool that programmatically displays information about data in a wire, and can determine whether to apply a breakpoint in order to pause execution. See also *Probe*.

D

D/A

Digital-to-analog conversion. The opposite operation of an A/D.

DAQ Assistant

An *Express VI* used for quickly configuring a data acquisition Task.

DAQmx Task

See *NI-DAQmx Task*.

Data Acquisition (DAQ)

Process of acquiring data, usually by performing an analog-to-digital (A/D) conversion. Its meaning is sometimes expanded to include data generation (D/A).

data dependency

Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. See also *Artificial Data Dependency*.

data 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.

database

Data that is organized for use by one or more computers. Usually, a database is a networked software application that provides controlled and efficient access to shared enterprise data.

dataflow

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.

datalog file

File that stores data as a sequence of records of a single, arbitrary data type that you specify when you create the file. Although all the records in a datalog file must be 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.

datasocket

A communications protocol supported by LabVIEW for sharing live data across a network.

DC

Direct current. The opposite of AC (alternating current). Refers to a very low frequency signal, such as one that varies less than once a second.

decoration

An object that is used on the front panel or block diagram in order to "decorate" (enhance the GUI) or document the VI. It serves no functional purpose. For example: lines, arrows, boxes, circles, and so on.

description property

Description of the front panel object that appears in the Context Help window when you move the cursor over the object and in VI documentation you generate.

device

A plug-in DAQ board.

device number

Number assigned to a device (DAQ board) in the NIDAQ configuration utility.

diagram disable structure

A structure that has one or more subdiagrams, or cases, of which only the Enabled subdiagram executes. It is commonly used to disable ("comment out") a section of the block diagram.

description box

Online documentation for a LabVIEW object.

destination terminal

See sink terminal.

dialog box

An interactive screen with prompts in which you specify additional information needed to complete a command.

differential measurement

Way to configure a device to read signals in which the inputs need not be connected to a reference ground. The measurement is made between two input channels.

digital data

Data that consists of 0's and 1's.

digital waveform

A special Waveform Datatype that contains Digital Data.

digital waveform graph

A graph that can display Digital Waveforms.

dimension

Size and structure attribute of an array.

DLL

Dynamically Linked Library. A Shared Library on the Windows operating system.

DMA

Direct memory access. A method by which you can transfer data to computer memory from a device or memory on the bus (or from computer memory to a device) while the processor does something else. DMA is the fastest method of transferring data to or from computer memory.

drag

The act of moving the mouse cursor on the screen to select, move, copy, or delete objects.

dynamic data

A special LabVIEW data type that adapts to just about any type of data that can be displayed in a graph.

Dynamically Linked Library

See DLL.

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.

enterprise

A business or organization.

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).

error cluster

A LabVIEW data structure that can contain information about errors that occur during program execution. See also *Error Handling*.

error handling

Taking action, in a software program, as a result of errors that occur during program execution.

event data node

The node on the left side of an Event Structure's event case, which contains event data. See also: *Event Structure*, *Event Filter Node*.

event filter node

The node on the right side of an Event Structure's event case, which is available only for *Filter Events* and is used for writing filtered event data or for discarding the event.

event structure

A LabVIEW structure used for capturing user interface and dynamic events.

event-driven programming

A style of programming that waits for events to occur (for example, using an *Event Structure*), rather than repeatedly testing (polling) for data conditions to change.

execution highlighting

Feature that animates VI execution to illustrate the dataflow in the VI.

expandable node

Refers to a subVI or function that can have a variable number of inputs and outputs at edit time. The number of inputs and outputs can be shown or hidden by resizing an Expandable Node vertically.

Express VI

A subVI designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.

expression node

A node that is similar to a *Formula Node*, but having only a one-line, single variable formula (expression).

F

feedback node

Similar to a *Shift Register*, but having parts that may be moved around inside a loop, rather than attached to the loop wall. Used to pass data from one iteration of a loop to the next.

FFT

Fast Fourier transform.

file marker

The current position in a file where read or write operations will occur, if not explicitly specified. The File Marker is automatically shifted, when file read and write operations are performed, by the number of bytes read or written. This is also referred to as the *File Position*.

file position

See *File Marker*.

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.

filter event

A special type of event that may be filtered programmatically inside an *Event Structure*. The data may be filtered or the event may be filtered, altogether. Filter Events will have an *Event Filter Node* on the right side of the event case.

find and replace

A LabVIEW feature that allows you to find objects and text, as well as replace the search result items with other objects and text.

flat sequence structure

A *Sequence Structure* whose subdiagram frames are organized from left to right, side-by-side. This is visually different, but functionally almost identical, to a *Stacked Sequence Structure*.

flattened data

Data of any type that have been converted to a string, usually for writing it to a file.

floating signal

A signal source in which the voltage signal is not referenced to any common ground, such as earth or building ground.

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 . . .
```

format and precision

The display properties of a numeric control or indicator. For example: scientific, engineering, decimal, number of decimal places, zero padding, and so on.

formatting string

A string that uses special formatting codes to define the format and precision of a numeric, for display or string representation.

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.

frequency

The rate at which something occursevents per unit time. See also *Hertz*.

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

Nonreentrant 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 sub-VIs is shared and thus can be used to pass global data among them.

glyph

A small picture or icon.

GOOP

Graphical Object-Oriented Programming. A pattern for object-oriented programming (OOP), in LabVIEW. See also OOP.

GPIB

General purpose interface bus. Also known as HP-IB (Hewlett-Packard Interface Bus) and IEEE 488.2 bus (Institute of Electrical and Electronic Engineers standard 488.2), it has become the world standard for almost any instrument to communicate with a computer. Originally developed by Hewlett-Packard in the 1960s to allow their instruments to be programmed in BASIC with a PC. Now IEEE has helped define this bus with strict hardware protocols that ensure uniformity across instrument.

graph control

Front panel object that displays data in a Cartesian plane.

graphical programming

Using graphical notation to program software applications. For example, programming in

LabVIEW (G). See also G.

ground

The common reference point in a system; i.e., ground is at 0 volts.

ground reference

System ground. See also *Grounded Signal*.

grounded signal

Signal sources with voltage signals that are referenced to a system ground, such as a building ground. Also called referenced signal sources.

PREV NEXT

H

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 the descriptions and data types of control attributes. The window also accesses LabVIEW's Online Reference.

hertz, Hz

Cycles per second.

hex

Hexadecimal. A base-16 number system.

hierarchical palette

Menu that contains palettes and subpalettes.

hierarchy

See *VI Hierarchy*.

hierarchy window

Window that graphically displays the hierarchy of VIs and subVIs.

history

See *VI History*.

housing

Nonmoving part of front panel controls and indicators that contains sliders and scales.

HTML

HyperText Markup Language. A markup language for creating web pages that may be viewed in a web browser.

HTTP

HyperText Transfer Protocol. A network protocol used for requesting and sending HyperText documents and data between a web browser (client) and web server (server).

PREV NEXT

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.

import picture

A LabVIEW feature that allows image files to be imported into the LabVIEW clipboard, in order to paste them onto front panels or block diagrams, or to replace image parts of controls and indicators (using the *Control Editor*).

indicator

Front panel object that displays output.

Inf

Digital display value for a floating-point representation of infinity. (-Inf represents negative infinity.)

INI File

A special file format that is commonly used for storing application configuration data and contains one or more named sections having key-value pairs. INI files are commonly referred to as *Configuration Files* or *Config Files*. INI files are usually saved with an .ini file extension.

instrument driver

VI that controls a programmable instrument.

instrument I/O assistant

An *Express VI* used for quickly configuring an instrument communication task.

intensity charts and graphs

Displays a 2D array of intensity data (an image). Color is used to represent the intensity value.

internet

The network of computers across the world (and beyond), which can communicate with each other using Internet Protocols such as *TCP-IP* and *UDP*.

invoke node

A node which is used to invoke a *Method* of an object.

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.

K

key focus

A front panel control which is currently the target ("focus") of keyboard inputs.

key-value pair

An element of an associative array. Commonly, the key is a string, and the value is any LabVIEW data type. For example, an *INI file* contains sections of key-value pairs.

L

label

Text object used to name or describe other objects or regions on the front panel or block diagram. See also *Name Label* and *Caption Label*.

labeling tool

Tool used to create labels and enter text into text windows.

LabVIEW

A graphical programming language and development tool from National Instruments. LabVIEW originally was an acronym for Laboratory Virtual Instrument Engineering Workbench.

LED

Light-emitting diode.

legend

Object owned by a chart or graph that displays the names and plot styles of plots on that chart or graph.

line

The equivalent of an analog channela path where a single digital signal is set or retrieved.

listbox

A LabVIEW control that contains a list of strings. The user can select one or more elements, reorder elements by *Drag & Drop*, and so on.

LLB File

Special LabVIEW file that contains a collection of related VIs (and other LabVIEW files) for a specific use. It is similar to a ZIP file that only LabVIEW can read.

local variable

A block diagram structure that can read from or write to the value of a control or indicator on the front panel (of the same VI).

PREV NEXT

M

marquee

A moving, dashed border that surrounds selected objects.

matrix

A mathematical structure that represents a two-dimensional array.

MAX

See *NI-MAX*.

mechanical action

A behavior of Boolean controls that defines when the value actually changes (as read by the block diagram) as a result of the user pressing (and possibly releasing) the Boolean. The mechanical action also defines when (if at all) the button rebounds after being read by the block diagram.

menu bar

Horizontal bar that contains names of main menus.

method

A function that belongs to an object that performs an action on the object. In LabVIEW, VI Server (and other) object methods are invoked using an *Invoke Node*.

modular programming

Programming that uses interchangeable computer routines.

multi-plot cursor

A plot cursor which has an adjustable X value and displays the Y values of one or more plots, at the specified X value.

PREV NEXT

N

name label

Label of a front panel object used to name the object and distinguish it from other objects. The label also appears on the block diagram terminal, local variables, and property nodes that are part of the object. See also *Caption Label*.

NaN

"Not A Number." Digital display value for a floating-point representation of not a number, typically the result of an undefined operation, such as division by zero.

network

A group of computers (and/or devices) that can communicate with each other.

network-published shared variables

A *Shared Variable* that has been configured to be accessible to other computers on the network. See also: *Shared Variable*, *NI-PSP*.

NI Example Finder

A LabVIEW utility for quickly finding LabVIEW examples.

NI-DAQ

Driver software for National Instruments DAQ boards and SCXI modules. This software acts as an interface between LabVIEW and the devices.

NI-DAQmx

The National Instruments data acquisition driver and application programming interface.

NI-DAQmx Task

An NI-DAQmx abstraction, containing a group of physical channels, along with timing and triggering information. Nearly all NI-DAQmx function calls operate on NI-DAQmx Tasks.

NI-MAX

National Instruments Measurement & Automation Explorer. A configuration utility that interacts with NI-DAQ, allowing you to configure your hardware, set up virtual channels, and test your I/O from the desktop.

NI-PSP

National Instruments Publish and Subscribe Protocol. A proprietary networking technology that provides fast and reliable data transmission for large and small applications. NI-PSP is used by network-published shared variables to communicate between VIs, remote computers, and hardware. See also *Shared Variable*.

nodes

Execution elements of a block diagram consisting of functions, structures, and subVIs.

nondisplayable characters

ASCII characters that cannot be displayed, such as new line, 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.

notifier

A LabVIEW messaging construct that is generally used to publish non-buffered data by one or more producers to one or more consumers. The Notifier Operations Functions are used to create and use of Notifiers.

notify event

An event which cannot be filtered. See also: *Event Structure*, *Filter Event*.

numeric controls and indicators

Front panel objects used to manipulate and display or input and output numeric data.

numeric representation

The type (signed integer, unsigned integer, floating point, complex number) and size (8 bit, 16 bit, 32 bit, 64 bit, etc.).

NRSE

Nonreferenced single-ended.

NRSE Measurement

All measurements are made with respect to a common reference. This reference voltage can vary with respect to ground.

nyquist frequency

One-half the sampling frequency. If the signal contains any frequencies above the Nyquist frequency, the resulting sampled signal will be aliased or distorted.

O

object

In LabVIEW, "object" is frequently used as a generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures; not to be confused with an object in the *OOP* sense.

object pop-up menu tool

Tool used to access an object's pop-up menu.

occurrence

A LabVIEW messaging construct that is purely event basedno data is associated with occurrences. The Occurrence Functions are used to create and use Occurrences.

octal

A base-eight numbering system.

OOP

Object-Oriented Programming. See also *GOOP*.

open source

Software which is licensed in a way that provides end users the right to view and modify the source code, distribute derivative works, and access the software for free. There are a variety of open source licenses. See also *OpenG*.

OpenG

An organized community of LabVIEW developers, found on the Web at www.openg.org, who collaboratively develop open source LabVIEW tools. See also *Open Source*.

operating tool

Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

optional input

A SubVI or function input that may be optionally wired (and having a standard default value). See also: *Recommended Input*, *Required Input*.

options dialog

The dialog where you can modify the LabVIEW options/preferences. This is opened by selecting the *Tools>>Options* menu item.

PREV NEXT

P

palette

Menu of pictures that represent possible options.

path

The location of a file within a file system. See also *Absolute Path*.

physical channel

Specifies a DAQ device and channel where a signal is physically connected.

picture control

A control used to display graphics that can be drawn using low-level picture function.

pipe

A special file that is used to read and write data between different applications on a computer. It can be thought of as a queue that is accessible across applications. Use the Pipe Functions to access pipes in LabVIEW.

platform

Computer and operating system.

platform dependent

Software components or functions that only work on specific operating systems. For example, *ActiveX* is platform dependentit only works on Windows.

plot

A graphical representation of an array of data shown either on a graph or a chart.

plot legend

A part of the graph/chart that displays the name, line color, and line style of the plots.

Polymorphic VI

A special VI type that links to multiple VIs that handle different datatypes for the same operation.

Polymorphic VI Selector

Part of a Polymorphic SubVI that allows the programmer to specify which polymorphic member VI to call. See also *Polymorphic VI*.

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 (Windows and Linux) or while holding down the command key (Mac OS X).

Pop-up Menus

Menus accessed by popping up, usually on an object. Menu options pertain to that object specifically.

port

A collection of digital lines that are configured in the same direction and can be used at the same time.

positioning tool

Tool used to move, select, and resize objects.

priority

Used to control the execution order of parallel tasks (VIs). If two VIs are waiting to execute, the one with higher priority VIs will usually execute first.

probe

Debugging feature for checking intermediate values in a VI.

probe tool

Tool used to create probes on wires.

profile window

Used to generate and view statistics about the execution of VIs. For example, number of times called, total execution time, average execution time, and so on. This is commonly used as a tool for optimizing code.

programmatic printing

Automatic printing of a VI front panel after execution.

project library

A collection of VIs that contain public and private member VIs. Public member VIs can be called by any other VIs, and private member VIs may only be called by members of the same library.

property

An attribute of an object. See also *Property Node*.

property node

A node used for reading or writing to an object's properties. See also *Property*.

pseudocode

Simplified language-independent representation of programming code.

Pull-down Menus

Menus accessed from a menu bar. Pull-down menu options are usually general in nature.

pulse

A signal whose amplitude deviates from zero (or some baseline level) for some (usually short) period of time.

PXI

PCI eXtensions for Instrumentation. A modular instrumentation platform based on Compact PCI with additional timing and triggering lines on the communication bus.

PREV NEXT

Q

queue

A LabVIEW messaging construct that is generally used to publish data by one producer to one or more consumers. The Queue Operations Functions are used to create and use of Queues.

queued message handler

A programming pattern that uses a *While Loop* with a *Case Structure* inside of it to process and execute messages that are stored in a queue. Each frame of the *Case Structure* handles a specific message and may operate on the message queue by enqueueing new messages, flushing the queue, and so on. Data is generally shared by means of one or more *Shift Registers*. See also *State Machine*.

R

race condition

Refers to an undesirable scenario, whereby two tasks (e.g., two While Loops) read from and write to shared data (e.g., a local variable), without appropriate synchronization, so that the state of the shared data depends on which tasks complete execution first. Often this results in shared data that gets overwritten and behavior that is non-deterministic.

radio buttons

A control having multiple Booleans, where only one of the Booleans may be TRUE.

radix

A display setting for integer numerics that shows the base system: Decimal, Hex, Octal, Binary...

read mode

A mode of structure or function (such as a Local Variable, Property Node, or Shared Variable) where it is configured to read a value. See also *Write Mode*.

recommended input

A SubVI or function input that is recommended to be wired. See also: *Required Input*, *Optional Input*.

reentrant execution

Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage. See also: *Reentrant VI*, *Clone*.

Reentrant VI

A VI that is configured for Reentrant Execution. See also *Reentrant Execution*.

regular expression

A special syntax for describing a set of (string) character sequences. Regular expressions are used by the Match Pattern and Match Regular Expression functions.

remote panel

A VI Front Panel that is being displayed on a (remote) computer other than the computer where the VI is actually executing.

rendezvous

A LabVIEW messaging construct that is generally used to synchronize parallel tasks, having them wait until all parallel tasks finish and "meet" to wait for each other at the rendezvous. The Rendezvous Functions are used to create and use Rendezvous.

Report Generation VIs

A set of LabVIEW VIs used for generating and printing reports.

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.

required input

A SubVI or function input that is required to be wired, otherwise the calling VI will be broken. See also: *Recommended Input*, *Optional Input*.

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.

RS-232

Recommended Standard #232. A standard proposed by the Instrument Society of America for serial communications. It's used interchangeably with the term "serial communication," although serial communications more generally refers to communicating one bit at a time. A few other standards you might see are RS-485, RS-422, and RS-423.

RSE

Referenced single-ended.

RSE Measurement

All measurements are made with respect to a common ground; also known as a grounded measurement.

PREV NEXT

S

sample

A single analog input or output data point.

sampling rate

The rate at which data points (samples) are being acquired (sampled). Has units of samples per second, or *Hertz*. See also: *Sample*, *Hertz*.

scalar

Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans 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.

scale legend

Part of the graph that is used to control the graph scale settings, such as autoscaling, scale locking, scale format, and so on.

scope mode

Mode of a waveform chart modeled on the operation of an oscilloscope.

SCPI

Standard Commands for Programmable Instrumentation, a standard that defines a set of commands to control programmable test and measurement devices. Many *instrument drivers* in LabVIEW implement SCPI.

scroll tool

Tool used to scroll windows.

SCXI

Signal Conditioning eXtensions for Instrumentation. A high-performance signal conditioning system devised by National Instruments, using an external chassis that contains I/O modules for signal conditioning, multiplexing, etc. The chassis is wired into a DAQ board in the PC.

secret of the universe

42.

selector terminal

The input terminal on a *Case Structure* that reads which Case should execute.

semaphore

A LabVIEW programming construct that protects access to shared resources (such as a subVI). Also known as a mutex. The Semaphore VIs are used to create, acquire, release, and destroy semaphores.

sequence local

Terminal that passes data between the frames of a Sequence Structure.

serial

Refers to the process of sending data one bit at a time, sequentially. Usually the term refers to an instrument that uses a serial communication protocol such as RS-232.

serial port

See also *RS-232*.

shared library

See also *Dynamically Linked Library (DLL)*.

shared variable

A data structure in LabVIEW that allows sharing data across a network, encapsulated in a single variable.

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. Sequence Structures come in two types: *Flat Sequence Structure* (good) and *Stacked Sequence Structure* (evil).

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. See also *Feedback Node*.

signal conditioning

Refers to the modifications done in hardware to a signal to make it easier to read; for example, amplification or noise reduction.

signal

Data that usually represents some physical process, such as temperature.

simulated device

A feature in *NI-MAX* that allows you to simulate National Instrument's hardware devices as if they were present on your computer.

single-ended measurement

A measurement mode in which one of the two inputs for an analog measurement is connected to ground or a common reference. This is in contrast to *differential measurement*.

single-step mode

Debugging mode in LabVIEW that allows you to execute the block diagram code one node at a time.

sink terminal

Terminal that absorbs data. Also called a destination terminal.

slider

Moveable part of slide controls and indicators.

source distribution

Refers to distributing, or including, your source code with your software.

source terminal

Terminal that emits data.

splitter bars

A tool in LabVIEW used to separate the front panel into multiple regions called panes. Each of these panes acts like a unique front panel.

spreadsheet file

A *Text File* that uses a comma, tab, or other character as the delimiter to separate fields. See also: *Text File*, *CSV File*.

stacked sequence structure

A *Sequence Structure* whose subdiagram frames are organized one on top of the other, similar to the cases of a Case Structure. This is different from a *Flat Sequence Structure*.

state machine

A method of execution in which individual tasks are separate cases in a Case Structure that is embedded in a While Loop. Sequences are specified as arrays of case strings. See also *Queued Message Handler*.

strict type Def.

A kind of Type Definition that forces all of its instance to preserve the same appearance and most properties of the type definition. See also *Type Definition*.

string

Text datatype.

string controls and indicators

Front panel objects used to manipulate and display or input and output text.

strip mode

Mode of a waveform chart 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.

subdiagram

Block diagram within the border of a structure.

subpalette

A palette contained within another palette. For example, the Structures palette is a subpalette of the Controls palette.

subpanel

A LabVIEW control that is a container that displays the front panel of another VI on the front panel of the current VI

SubVI

VI used in the block diagram of another VI; comparable to a subroutine.

sweep mode

Similar to scope modeexcept a line sweeps across the display to separate old data from new data.

system controls

These are controls and indicators designed specifically for use in dialog boxes, such as rings, radio buttons, checkboxes, and so on. They preserve the look and feel of generic software applications seen on the operating system,

system Exec

A function in LabVIEW that sends a system command, such as one you might type into a command prompt.

PREV NEXT

T

tab control

A LabVIEW control that has multiple pages that are accessed by clicking on the named tab that corresponds to the desired page.

table

A LabVIEW control that can display a 2D array of strings. It also allows the user to interactively edit the values of the string elements.

task

See *NI-DAQmx Task*.

TCP/IP

Transmission Control Protocol/Internet Protocol. A connection-based protocol that is the underlying protocol for the Internet and most internal networks. See also *UDP*.

terminal

Object or region on a node through which data pass.

text file

See also *Binary File*.

time stamp control

A LabVIEW control that stores an absolute date and time.

timed loop

A LabVIEW structure, similar to a *While Loop*, that executes one or more subdiagrams, or frames, sequentially in each iteration of the loop at the period you specify. See also *Timed Structure*.

timed sequence

A LabVIEW structure, similar to a *Flat Sequence Structure*, that executes one or more subdiagrams, or frames, sequentially at the time you specify. See also *Timed Structure*.

timed structure

A set of LabVIEW structures that allow you to control the rate and priority at which they execute their subdiagrams, as well as control their synchronization using timing sources. See also: *Timed Loop*, *Timed Sequence*.

tip strip

Text that is displayed when the mouse cursor is hovered over a control or indicator. Often used to provide a more detailed description of a control or indicator's function.

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.

toolkit

A LabVIEW add-on component that adds VIs to your palette that perform a specific function.

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.

tree control

A LabVIEW control that displays a hierarchical list of items that the user can interact with by selecting items, dragging and dropping items, expanding and contracting items, and so on.

trigger

A condition for starting or stopping a DAQ operation.

tunnel

Data entry or exit terminal on a structure.

type definition (typedef)

See also *Strict Type Def*.

typecast

To change the type descriptor of a data element without altering the memory image of the data.

type descriptor

See *Data Type Descriptor*.

U

UDP

User Datagram Protocol. A "connection-less" protocol that is similar to TCP in that it relies on IP addresses, except that the sender of messages in not aware of listeners. See also *TCP*.

unbundle node

Function that separates (unbundles) a cluster into one or more individual elements. See also *Unbundle Node*.

unit

A unit of measurement. A physical dimension. For example, meter, foot, year, deg C, Joule, millisecond.

universal constant

Uneditable block diagram object that emits a particular ASCII character or standard numeric constant; for example, pi.

URL

Uniform Resource Locator. A logical address that identifies a resource on a server, usually on the Web. For example, http://www.labviewforeveryone.com/ is the URL for the web site of this book.

USB

Universal Serial Bus. A very common serial bus for connecting devices, usually computer peripherals.

user-defined constant

Block diagram object that emits a value you set.

V

variant

A generic LabVIEW data type that can accept any other LabVIEW datatype. The data of a variant, is both the data and type information of data type that it "wraps".

VI

See virtual instrument.

VI Description

A documentation string that describes the functionality of a VI. It is intended to help LabVIEW programmer understand how a VI works.

VI Hierarchy

The entire set of all VIs that are subVIs (recursively) of a Top-Level VI.

VI History

The history of comments, if provided, for all revisions to a VI.

VI Property

A VI Server Property, which is accessible via a Property Node, for the VI class. See also: VI Server, Property Node.

VI Server

A feature in LabVIEW that allows you to programmatically and remotely control the behavior of VIs and controls.

virtual channel

Defines a real-world measurement consisting of one or more DAQ channels (terminals on your DAQ device) along with other channel specific informationrange, terminal configuration, and custom scalingthat is used to format the data.

virtual instrument

LabVIEW program; so called because it models the appearance and function of a physical instrument.

VISA

Virtual Instrument Standard Architecture. A driver software framework that unifies instrumentation software standards, whether the instrument uses GPIB, PXI, VXI, or Serial (RS-232/422/485).

VISA Resource

A string, used by the VISA framework, that specifies the physical address of a device.

VXI

VME eXtensions for Instrumentation. A very high-performance system for instrumentation.

PREV NEXT

W

waveform

A data type in LabVIEW that usually represents an analog signal; it bundles the Y-axis data with the timing information (Xo and delta-X).

web server

An application that uses the *HTTP* protocol to serve web pages (*HTML* documents) to web browsers and other client applications.

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.

write mode

A mode of structure or function (such as a Local Variable, Property Node, Shared Variable, and so on) where it is configured to write a value. See also *Read Mode*.

X

XML

eXtensible Markup Language. A set of rules for formatting data as text, allowing the data to be shared across systems in a format that is independent of both programming language and platform.

XY Graph

A general-purpose, Cartesian graphing object that plots multivalued functions, such as circular shapes or waveforms with a varying time base. The XY graph displays any set of points, evenly sampled or not.

Index

Index

Index

Index

Index

Index

Index

Index

PREV NEXT

Index

PREV NEXT

Index

Index

Index

Index

PREV NEXT

Index

Index

Index

PREV NEXT

Index

PREV NEXT

Index

Index

Index

Index

Index

Index

Index

Index

Index