LabVIEW™

User Manual

NATIONAL INSTRUMENTS™
The Software is the Instrument™

January 1998 Edition
Part Number 320999B-01

Internet Support
E-mail: support@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://www.natinst.com

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00,
Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

CVI™, LabVIEW™, National Instruments™, natinst.com™, NI-488™, NI-488.2™, NI-DAQ™, NI-VISA™, NI-VXI™, SCXI™, and VXIpc™, are trademarks of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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

Contents

About This Manual

Chapter 1
Introduction

PART I
Introduction to G Programming

Chapter 2
Creating VIs

Chapter 3
Loops and Charts

Chapter 4
Case and Sequence Structures and the Formula Node

Chapter 5
Arrays, Clusters, and Graphs

Chapter 6
Strings and File I/O

PART II
I/O Interfaces

Chapter 7
Getting Started with a LabVIEW Instrument Driver

Chapter 8
LabVIEW VISA Tutorial

Chapter 9
Introduction to LabVIEW GPIB Functions

Chapter 10
Serial Port VIs

PART III
Analysis

Chapter 11
Introduction to Analysis in LabVIEW

Chapter 12
Signal Generation

Chapter 13
Digital Signal Processing

Chapter 14
Smoothing Windows

Chapter 15
Spectrum Analysis and Measurement

Chapter 16
Filtering

Chapter 17
Curve Fitting

Chapter 18
Linear Algebra

Chapter 19
Probability and Statistics

Part IV
Network and Interapplication Communication

Chapter 20
Introduction to Communication

Chapter 21
TCP and UDP

Chapter 22
ActiveX Support

Chapter 23
Using DDE

Chapter 24
AppleEvents

Chapter 25
Program-to-Program Communication

Part V
Advanced G Programming

Chapter 26
Customizing VIs

Chapter 27
Front Panel Object Attributes

Chapter 28
Program Design

Chapter 29
Where to Go from Here

Appendices, Glossary, and Index

Figures, Tables, and Activities

Figures

Tables

Activities

About This Manual

The *LabVIEW User Manual* provides information about creating virtual instruments (VIs). This manual also includes information about the interfaces to which you can input and output data, using LabVIEW VIs to perform analysis operations, and how LabVIEW handles network and interapplication communication. Please read the *LabVIEW Release Notes* before you use the *LabVIEW User Manual*.

Organization of This Manual

The *LabVIEW User Manual* is organized as follows.

- Chapter 1, *Introduction*, introduces the unique LabVIEW approach to programming. It also explains how to start using LabVIEW to develop programs.

Part I, Introduction to G Programming

This section contains basic information about creating virtual instruments (VIs), using VIs in other VIs, programming structures such as loops, and data structures such as arrays and strings.

Part I, *Introduction to G Programming*, contains the following chapters.

- Chapter 2, *Creating VIs*, explains how to create a VI including the front panel, which is the user interface, and the block diagram, which is the source code. Once you create a VI, you can use it in other VIs.
- Chapter 3, *Loops and Charts*, shows you how to repeat portions of the block diagram using a While Loop and a For Loop. This chapter also explains how to display graphically multiple points, one at a time, on a chart.
- Chapter 4, *Case and Sequence Structures and the Formula Node*, explains how to use the Case structure, which is a conditional structure, the Sequence structure, which aids in establishing execution order, and the Formula Node, which aids in executing mathematical formulas.
- Chapter 5, *Arrays, Clusters, and Graphs*, shows how to display a group or array of data points on a graph. You can pass scale parameters as well as an array of data points to a graph by creating a cluster, which is a group of data different data types.
- Chapter 6, *Strings and File I/O*, introduces string controls and indicators and file input and output operations.

Part II, I/O Interfaces

This section contains basic information on the interfaces to which you can input and output data, which are data acquisition, GPIB, serial, and VXI. Refer to the *Data Acquisition Basics Manual* for basic information on real-time data acquisition. VISA (Virtual Instrument Software Architecture) is a single software library that interfaces with GPIB, serial, and VXI instruments. LabVIEW applications developed especially for a specific instrument are called instrument drivers. National Instruments provides several instrument drivers using the VISA library, but you can also build your own instrument drivers.

Part II, *I/O Interfaces*, contains the following chapters.

- Chapter 7, *Getting Started with a LabVIEW Instrument Driver*, explains how to create and use National Instruments instrument drivers.
- Chapter 8, *LabVIEW VISA Tutorial*, shows you how to implement common VISA applications using message-based and register-based communication as well as events and locking.
- Chapter 9, *Introduction to LabVIEW GPIB Functions*, explains how the GPIB operates and the difference between the IEEE 488 and IEEE 488.2 interface.
- Chapter 10, *Serial Port VIs*, explains the important factors that affect serial communication.

Part III, Analysis

This section contains basic information on analysis of data, signal processing, signal generation, linear algebra, curve fitting, probability, and statistics.

Part III, *Analysis*, contains the following chapters.

- Chapter 11, *Introduction to Analysis in LabVIEW*, introduces concepts that apply to all analysis applications, including supported functionality, notation and naming conventions, and sampling signal methods.
- Chapter 12, *Signal Generation*, explains how to produce signals using the normalized frequency and how to build a simulated function generator.
- Chapter 13, *Digital Signal Processing*, shows the difference between the Fast Fourier Transform (FFT) and the Discrete Fourier Transform (DFT).

- Chapter 14, *Smoothing Windows*, explains how using windows prevents spectral leakage and improves the analysis of acquired signals.
- Chapter 15, *Spectrum Analysis and Measurement*, shows how to determine the amplitude and phase spectrum, develop a spectrum analyzer, and determine the total harmonic distortion (THD).
- Chapter 16, *Filtering*, explains how to filter unnecessary frequencies from signals using infinite impulse response filters (IIR), finite impulse response filters (FIR), and nonlinear filters.
- Chapter 17, *Curve Fitting*, shows how to extract information from a data set to create a data trend description.
- Chapter 18, *Linear Algebra*, explains how to perform matrix computation and analysis.
- Chapter 19, *Probability and Statistics*, explains some fundamental concepts of probability and statistics, and shows how to use these concepts in solving real-world problems.

Part IV, Network and Interapplication Communication

This section contains basic information about network and interapplication communication.

Part IV, *Network and Interapplication Communication*, contains the following chapters.

- Chapter 20, *Introduction to Communication*, introduces the way LabVIEW handles networking and interapplication communication.
- Chapter 21, *TCP and UDP*, explains basic concepts of Transmission Control Protocol (TCP), Internet Protocol (IP), and internet addresses.
- Chapter 22, *ActiveX Support*, shows how LabVIEW can be an ActiveX server and client. ActiveX is the same as OLE Automation communication.
- Chapter 23, *Using DDE*, explains how to use Dynamic Data Exchange (DDE) to communicate between Windows applications. DDE can be used in a client, a server, and across a network.
- Chapter 24, *AppleEvents*, shows how AppleEvents are used to communicate between LabVIEW and other Macintosh applications. LabVIEW can be an AppleEvents server and client.
- Chapter 25, *Program-to-Program Communication*, explains how LabVIEW can communicate to other Macintosh applications using Program-to-Program Communication (PPC).

Part V, Advanced G Programming

This section contains information on VI customization; programmatic control of front panel objects, VIs, and LabVIEW; and tips on how to design complex applications.

Part V, *Advanced G Programming*, contains the following chapters.

- Chapter 26, *Customizing VIs*, shows how to use **VI Setup…** and **VI Node Setup…** to customize the appearance and execution behavior of a VI when it is running.
- Chapter 27, *Front Panel Object Attributes*, describes objects called attribute nodes, which are special block diagram nodes that control the appearance and functional characteristics of controls and indicators.
- Chapter 28, *Program Design*, explains techniques to use when creating programs and offers programming-style guidelines.
- Chapter 29, *Where to Go from Here*, provides information about resources you can use to create your applications successfully.

Appendices, Glossary, and Index

- Appendix A, *Analysis References*, lists the reference material used to produce the Analysis VIs in LabVIEW. These references contain more information on the theories and algorithms implemented in the analysis library.
- Appendix B, *Common Questions*, answers common questions about LabVIEW networking communications and Instrument I/O, specifically GPIB and serial I/O.
- Appendix C, *Customer Communication*, contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation.
- The *Glossary* contains an alphabetical list of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

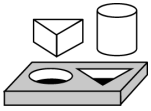
Conventions Used in This Manual

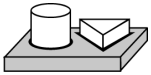
The following conventions are used in this manual:

< >
: Angle brackets enclose the name of a key on the keyboard—for example, <shift>. Angle brackets containing numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, DBIO<3..0>.

-
: A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.

»
: The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options» Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.


: This icon to the left of bold text denotes the beginning of an activity, which contains step-by-step instructions you can follow to learn more about LabVIEW.


: This icon to the left of bold text denotes the end of an activity, which contains step-by-step instructions you can follow to learn more about LabVIEW.


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


: This icon to the left of bold italicized text denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

bold
: Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs.

bold italic
: Bold italic text denotes an activity objective, note, caution, or warning.

`bold monospace`
: Bold monospace text denotes messages and responses that the computer automatically prints to the screen.

italic
: Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.

`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for statements and comments taken from programs.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.
Platform	Text in this font denotes information related to a specific platform.

Related Documentation

- *G Programming Reference Manual*
- *LabVIEW Data Acquisition Basics Manual*
- *LabVIEW Function and VI Reference Manual*
- *LabVIEW QuickStart Guide*
- LabVIEW *Online Reference*, available by selecting **Help»Online Reference**
- *LabVIEW Online Tutorial* **(Windows only)**, which you launch from the LabVIEW dialog box
- *G Programming Quick Reference Card*
- *LabVIEW Getting Started Card*
- *LabVIEW Release Notes*
- *LabVIEW Upgrade Notes*

Customer Communication

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

1

Introduction

This chapter introduces the unique LabVIEW approach to programming. It also explains how to start using LabVIEW to develop programs. The chapter refers you to other chapters or manuals for more information.

What Is LabVIEW?

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

LabVIEW, like C or BASIC, is a general-purpose programming system with extensive libraries of functions for any programming task. LabVIEW includes libraries for data acquisition, GPIB and serial instrument control, data analysis, data presentation, and data storage. LabVIEW also includes conventional program development tools, so you can set breakpoints, animate the execution to see how data passes through the program, and single-step through the program to make debugging and program development easier.

How Does LabVIEW Work?

LabVIEW is a general-purpose programming system, but it also includes libraries of functions and development tools designed specifically for data acquisition and instrument control. LabVIEW programs are called *virtual instruments* (*VIs*) because their appearance and operation can imitate actual instruments. However, VIs are similar to the functions of conventional language programs.

A VI consists of an interactive user interface, a dataflow diagram that serves as the source code, and icon connections that allow the VI to be

called from higher level VIs. More specifically, VIs are structured as follows:

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

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

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

G Programming

G is the easy to use graphical data flow programming language on which LabVIEW is based. G simplifies scientific computation, process monitoring and control, and test and measurement applications, and you also can use it for a wide variety of other applications.

Part I, *Introduction to G Programming*, covers the functionality of G that you need to get started with most LabVIEW applications. For a more extensive explanation of LabVIEW functionality, see the *G Programming Reference Manual*.

The basic concepts of G that are covered in this manual are described in the following list.

- VIs—Virtual instruments (VIs) have three main parts: the front panel, the block diagram, and the icon/connector. The front panel specifies the user interface of the VI. The block diagram consists of the executable code that you create using nodes, terminals, and wires. With the icon/connector, you can use a VI as a subVI in the block diagram of another VI. For more information about VIs, refer to Chapter 2, *Creating VIs* and Chapter 26, *Customizing VIs*.
- Loops and Charts—G has two structures to repeat execution of a sub-diagram—the *While Loop* and the *For Loop*. Both structures are resizable boxes. You place the subdiagram to be repeated inside the border of the loop structure. The While Loop executes as long as the value at the conditional terminal is TRUE. The For Loop executes a set number of times. Charts are used to display real-time trend information to the operator. For more information about loops and charts, refer to Chapter 3, *Loops and Charts*.
- Case and Sequence Structures—The *Case structure* is a conditional branching control structure, which executes a subdiagram based on certain input. A *Sequence structure* is a program control structure that executes its subdiagrams in numeric order. For more information about Case or Sequence structures, refer to Chapter 4, *Case and Sequence Structures and the Formula Node*.
- Attribute Nodes—*Attribute nodes* are special block diagram nodes that you can use to control the appearance and functional characteristics of controls and indicators. For more information about attribute nodes, refer to Chapter 27, *Front Panel Object Attributes*.
- Arrays, Clusters and Graphs—An *array* is a resizable collection of data elements of the same type. A *cluster* is a statically sized collection of data elements of the same or different types. Graphs commonly are used to display data. For more information about arrays, clusters, and graphs, refer to Chapter 5, *Arrays, Clusters, and Graphs*.

Organization of the LabVIEW System (Windows)

After you have completed the installation, as described in the *LabVIEW Release Notes* that come with your software, your LabVIEW directory should contain the following files.

- `LABVIEW.EXE`—This is the LabVIEW program. Launch this program to start LabVIEW.
- `vi.lib` directory—Contains libraries of VIs that are included with LabVIEW, including GPIB, analysis, and data acquisition (DAQ) VIs. Most of these are available from the **Functions** palette.
- `examples` directory—Contains numerous subdirectories of examples. This directory also contains a VI called `readme.vi` that serves as a guide to the examples.
- `serpdrv` and `daqdrv`—These files serve as part of LabVIEW's interface to the serial port, and DAQ communication, respectively. These files must be in the same directory as `vi.lib`.
- `resource` directory
  - `labview.rsc`, `lvstring.rsc`, and `lvicon.rsc`—Data files used by the LabVIEW application
  - **(Windows 3.1)** `lvdevice.dll`—This file provides timing services to LabVIEW and must be in the same directory as `vi.lib` for LabVIEW to run.
  - **(Windows 3.1)** `lvimage.dll`—This file allows LabVIEW to load images created using a variety of graphics programs.
  - `labview50.tlb`—This file is a type library to enable LabVIEW to act as an ActiveX server.
  - `ole_container.dll`—This file enables LabVIEW to display and update ActiveX containers.
  - `lvwutil32.dll`—This file is used by the Solution Wizard, which builds DAQ and Instrument I/O examples based on your criteria.
  - `lvjpeg.dll` and `lvpng.dll`—These files provide support to display JPEG and PNG graphics in HTML files when you print VI documentation to an HTML file.
- `Cintools` directory—Contains files necessary to build Code Interface Nodes (CINs), which are a means to link C code to LabVIEW VIs.
- `visarc` file—Serves as part of LabVIEW's interface to VISA (Virtual Instrument Software Architecture). VISA provides a single interface library for controlling VXI, GPIB, and Serial instruments.

- `labview.ini`—Contains the configuration options for LabVIEW.
- `Project` directory—Contains files which become items in the LabVIEW **Project** menu.
- `menus` directory—Contains files used to configure the structure of the **Controls** and **Functions** palettes.
- `Instr.lib` directory—Contains instrument drivers used to control VXI, GPIB, and Serial instruments. When you install National Instruments instrument drivers, place them in this directory because they will be added to the **Functions** palette.
- `Help` directory—Contains complete online documentation as well as the Search Examples help file, which aids in locating examples common to your application.
- `Tutorial` directory—Contains files that are necessary to run the online tutorial, an interactive tutorial covering the basic concepts of the LabVIEW environment.
- `Activity` directory—Is a location where you can save the VIs you create while completing the activities in this manual.
- `User.lib` directory—Is a location where you can save commonly used VIs that you have created. The VIs in this directory will be displayed in the **Functions** palette.
- `Wizard` directory—This directory creates the **Solution Wizard** option in the **File** menu. You can use this directory to add items to the **File** menu.

LabVIEW installs driver software for GPIB, data acquisition, and VXI driver hardware. For configuration information, see Chapter 2, *Installing and Configuring Your Data Acquisition Hardware*, in the *LabVIEW Data Acquisition Basics Manual*, the *VXI VI Reference Manual*, and Chapter 8, *LabVIEW VISA Tutorial*, of this manual.

Startup Screen on Windows

When you launch LabVIEW, you are greeted with a navigation dialog box where introductory material, common commands, and Quick Tips are easily accessible. If you prefer to bypass the navigation dialog, you can disable it using a checkbox at the bottom of the dialog box. To reenable it, use the Preferences dialog box.

When all VIs are closed, a similar dialog box appears. The **Small Dialog** button switches to a simpler version of the dialog box—with only **New**, **Open**, and **Exit** buttons.

Organization of the LabVIEW System (Macintosh)

After you have completed the installation, as described in the *LabVIEW Release Notes* that come with your software, your LabVIEW directory should contain the following files.

- `LabVIEW`—This is the LabVIEW program. Launch this program to start LabVIEW.
- `vi.lib` folder—Contains libraries of VIs that are included with LabVIEW, including GPIB, analysis, and data acquisition (DAQ) VIs. Most of these are available from the **Functions** palette.
- `examples` folder—Contains numerous subfolders of examples. This folder also contains a VI called `readme.vi` that serves as a guide to the examples.
- `resource` folder
  - `lvstring.rsrc` and `lvicon.rsrc`—Data files used by the LabVIEW application.
  - `lvjpeg.lib` and `lvpng.lib`—These files provide support to display JPEG and PNG graphics in HTML files when you print VI documentation to an HTML file.
- `cintools` folder—Contains files necessary to build Code Interface Nodes (CINs), which are a means to link C code to LabVIEW VIs.
- `visarc` file—Serves as part of LabVIEW's interface to VISA, Virtual Instrument Software Architecture. VISA provides a single interface library for controlling VXI, GPIB, and Serial instruments.
- `Project` folder—Contains files which become items in the LabVIEW **Project** menu.
- `menus` folder—Contains files used to configure the structure of the **Controls** and **Functions** palettes.
- `instr.lib` folder—Contains instrument driers used to control VXI, GPIB, and Serial instruments. When you install National Instruments instrument drivers, place them in this directory because they will be added to the **Functions** palette.
- `help` folder—Contains complete online documentation as well as the Search Examples help file, which aids in locating examples common to your application.
- `activity` folder—Is a location where you can save the VIs you create while completing the activities in this manual.

- `user.lib` folder—Is a location where you can save commonly used VIs that you have created. The VIs in this directory will be displayed in the **Functions** palette.
- `Wizard` folder—This directory creates the **Solution Wizard** option in the **File** menu (**PCI Macintosh only**). You can use this directory to add items to the **File** menu.

In addition, the LabVIEW installation utility installs several driver files so that you can use GPIB and/or DAQ plug-in boards.

- `System Folder:Control Panels:NI-488 INIT`—This control panel contains the drivers for your GPIB boards. You can use it to configure your boards, but you rarely need to change any settings.
- `System Folder:Control Panels:NI-DAQ`—This control panel loads DAQ drivers into memory. You can use it to configure the location and behavior of your DAQ boards and SCXI modules.
- `System Folder:Extensions:NI-DMA/DSP`—Both the GPIB and DAQ drivers use this extension. It provides support for direct memory access (DMA) transfer of data, which provides higher data transfer rates. This extension also provides support for NI-DSP boards.

LabVIEW installs driver software for GPIB and data acquisition hardware. For configuration information, see Chapter 2, *Installing and Configuring Your Data Acquisition Hardware*, in the *LabVIEW Data Acquisition Basics Manual*.

Organization of the LabVIEW System (UNIX)

After you have completed the installation, as described in the *LabVIEW Release Notes* that come with your software, your LabVIEW directory should contain the following files.

- `labview`—This is the LabVIEW program. Launch this program to start LabVIEW.
- `vi.lib` directory—Contains libraries of VIs that are included with LabVIEW, including GPIB, analysis, and data acquisition (DAQ) VIs. Most of these are available from the **Functions** palette.
- `examples` directory—Contains numerous subdirectories of examples. This directory also contains a VI called `readme.vi` that serves as a guide to the examples.
- `serpdrv`—This file serves as part of LabVIEW's interface to serial port communication. This file must be in the same directory as `vi.lib`.

- `resource` directory
  - `labview.rsc`, `lvstring.rsc`, and `lvicon.rsc`—Data files used by the LabVIEW application
  - `lvjpeg.lib` and `lvpng.lib`—These files provide support to display JPEG and PNG graphics in HTML files when you print VI documentation to an HTML file.
- `cintools` directory—Contains files necessary to build Code Interface Nodes (CINs), which are a means to link C code to LabVIEW VIs.
- `visarc` file—Serves as part of LabVIEW's interface to VISA, Virtual Instrument Software Architecture. VISA provides a single interface library for controlling VXI, GPIB, and Serial instruments.
- `Project` directory—Contains the files which become items in the LabVIEW **Project** menu.
- `menus` directory—Contains files used to configure the structure of the **Controls** and **Functions** palettes.
- `instr.lib` directory—Contains instrument drivers used to control VXI, GPIB, and Serial instruments. When you install National Instruments instrument drivers, place them in this directory because they will be added to the **Functions** palette.
- `help` directory—Contains complete online documentation as well as the Search Examples help file, which aids in locating examples common to your application.
- `activity` directory—Is a location where you can save the VIs you create while completing the activities in this manual.
- `user.lib` directory—Is a location where you can save commonly used VIs that you have created. The VIs in this directory will be displayed in the **Functions** palette.
- `Wizard` directory—This directory creates the **Solution Wizard** option in the **File** menu. You can use this directory to add items to the **File** menu.
- `acrobat` directory—Contains online documentation in Acrobat (`.pdf`) format.
- `acroread` directory—Contains Adobe Acrobat reader files.

Toolkit Support

Files that are installed in `vi.lib\addons` automatically show up at the top level of the **Controls** and **Functions** palettes. This feature can be used by new toolkits to make them more accessible after installation. If you already have toolkits that installed files elsewhere, you can move them to the `addons` directory for easier access. If you want to add your own VIs to the palettes, we recommend placing them in user.lib or adding them to a custom palette set.

Where Should I Start?

This manual provides basic information on how to build an application in LabVIEW. To become familiar with the LabVIEW environment, go through the *LabVIEW Online Tutorial* **(Windows only)**, the *LabVIEW QuickStart Guide*, and Part I, *Introduction to G Programming* in this manual.

Most LabVIEW applications are divided into the following tasks: I/O interface to sensors or instruments, data display on the front panel, data analysis, data storage, and data transfer across a network. To learn more about each of these tasks, refer to Part II, *I/O Interfaces*, Part III, *Analysis*, and Part IV, *Network and Interapplication Communication*. For advanced G programming techniques, refer to Part V, *Advanced G Programming*, in this manual.

To generate or find examples similar to your application, refer to the Solution Wizard **(Windows and PCI Macintosh only)** or Search Examples online help file **(Windows only)**, which you can access from the LabVIEW startup dialog.

For information on individual functions and VIs, refer to the *LabVIEW Function and VI Reference Manual* and online help.

Part I

Introduction to G Programming

This section contains basic information about creating virtual instruments (VIs), using VIs in other VIs, programming structures such as loops, and data structures such as arrays and strings.

Part I, *Introduction to G Programming*, contains the following chapters.

- Chapter 2, *Creating VIs*, explains how to create a VI including the front panel, which is the user interface, and the block diagram, which is the source code. Once you create a VI, you can use it in other VIs.
- Chapter 3, *Loops and Charts*, shows you how to repeat portions of the block diagram using a While Loop and a For Loop. This chapter also explains how to display multiple points graphically, one at a time, on a chart.
- Chapter 4, *Case and Sequence Structures and the Formula Node*, explains how to use the Case structure, which is a conditional structure, the Sequence structure, which aids in establishing execution order, and the Formula Node, which aids in executing mathematical formulas.
- Chapter 5, *Arrays, Clusters, and Graphs*, shows how to display a group or array of data points on a graph. You can pass scale parameters as well as an array of data points to a graph by creating a cluster, which is a group of different data types.
- Chapter 6, *Strings and File I/O*, explains how to manipulate strings and write those strings to an ASCII file.

Note **(Windows 3.1)** ***You must save the VIs you create in Part I in VI libraries. VI libraries allow you to use file names that are longer than 8 characters. Also, the VIs needed for the activities in Part I are located in the VI library*** `LabVIEW\Activity\Activity.llb`***. Refer to the*** *Saving VIs* ***section in Chapter 2****, Editing VIs****, of the*** *G Programming Reference Manual* ***for more information on VI Libraries.***

2

Creating VIs

This chapter introduces the basic concepts of virtual instruments and provides activities that explain the following:

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

What is a Virtual Instrument?

A virtual instrument (VI) is a program in the graphical programming language G. Virtual instrument front panels often have a user interface similar to physical instruments. G also has built-in functions that are similar to VIs, but do not have front panels or block diagrams as VIs do. Function icons always have a yellow background.

How Do You Build a VI?

One of the keys to creating LabVIEW applications is understanding and using the hierarchical nature of the VI. After you create a VI, you can use it as a subVI in the block diagram of a higher-level VI.

VI Hierarchy

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

As with other applications, you can save your VI to a file in a regular directory. With G, you also can save multiple VIs in a single file called a VI library.

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

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

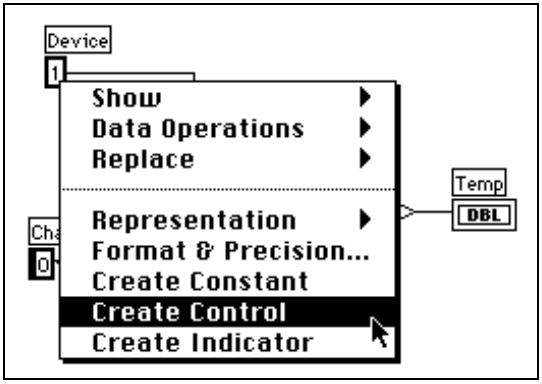
VI libraries have the same load, save, and open capabilities as other directories. VI libraries, however, are not hierarchical. That is, you cannot create a VI library inside of another VI library. You cannot create a new directory inside a VI library, either. There is no way to list the VIs in a VI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

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

Even though you might not save your own VIs in VI libraries, you should be familiar with how they work. In the various activities in this manual, you will save your VIs in the `LabVIEW\Activity` directory. Solutions to these activities are provided in the `LabVIEW\Activity\Solution` directory.

Controls, Constants, and Indicators

A control is an object you place on your front panel for entering data into a VI interactively or into a subVI programmatically. An indicator is an object you place on your front panel for displaying output. Controls and indicators in G are similar to input and output parameters, respectively, in traditional programming languages. An alternative to placing controls and indicators on the front panel and then wiring them to functions or VIs on the block diagram, is to create controls or indicators directly from the block diagram. To do this, *pop up* on the input terminal of a function or VI on the block diagram and select **Create Control**. This creates a control of the correct data type and wires it to the terminal.



You can create an indicator and wire it to an output terminal by popping up on the terminal and selecting **Create Indicator**. As an alternative to placing constants on the block diagram and wiring them to functions and VIs, you can pop up on a function or VI terminal and select **Create Constant**. You cannot delete a control or indicator from the block diagram. As with all front panel objects, you must go to the front panel, select the Positioning tool, and then delete the object.

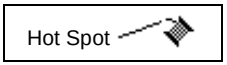
Each time you create a new control or indicator on the front panel, LabVIEW creates the corresponding terminal in the block diagram. The terminal symbols suggest the data type of the control or indicator. For example, a DBL terminal represents a double-precision, floating-point number; a TF terminal is a Boolean; an I16 terminal represents a regular, 16-bit integer; and an ABC terminal represents a string. For more information about data types in G, and their graphical representations, see the *G Programming Quick Reference Card*.

Terminals

Terminals are regions on a VI or function through which data passes. Terminals are analogous to parameters in text-based programming languages. It is important that you wire the correct terminals of a function or VI. You can view the icon connector to make correct wiring easier. To do this, pop up on the function or VI and choose **Show»Terminals**. To return to the icon, pop up on the function or VI and select **Show»Terminals** again.

Wires

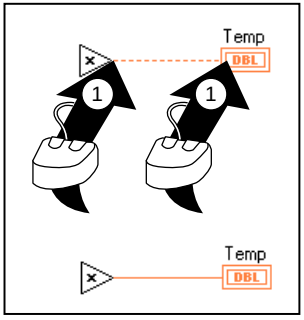
A *wire* is a data path between nodes. Wires are colored according to the kind of data each wire carries. Blue wires carry integers, orange wires carry floating-point numbers, green wires carry Booleans, and pink wires carry strings. For more information about wire styles and colors, see the *G Programming Quick Reference Card*.



To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and click on the second terminal. It does not matter at which terminal you start. The hot spot of the Wiring tool is the tip of the unwound wiring segment.

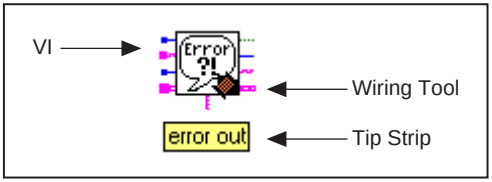
In the wiring illustrations in this section, the arrow at the end of this mouse symbol shows where to click and the number printed on the arrow indicates how many times to click the mouse button.

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



Tip Strips

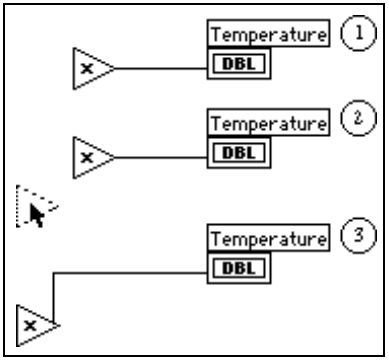
When you move the Wiring tool over the terminal of a node, a *tip strip* for that terminal pops up. Tip strips consist of small, yellow text banners that display the name of each terminal. These tip strips should help you to wire the terminals. The following illustration displays the tip strip that appears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



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

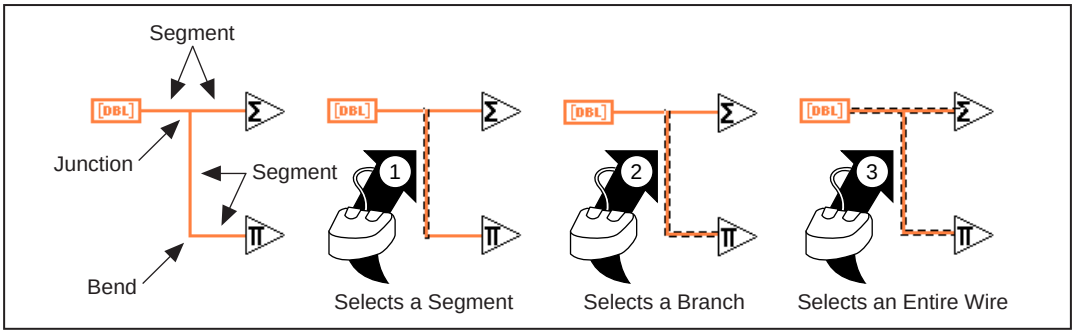
Wire Stretching

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



Selecting and Deleting Wires

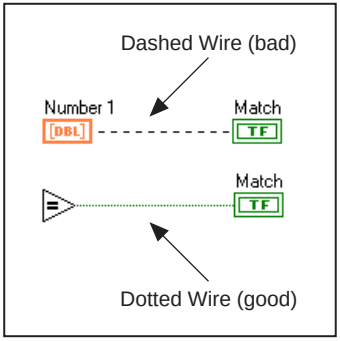
You might wire nodes incorrectly. If you do, select the wire you want to delete and then press <Delete>. A wire segment is a single horizontal or vertical piece of wire. The point where three or four wire segments join is called a *junction*. A wire branch contains all the wire segments from one junction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

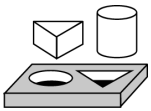


Bad Wires

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

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





Activity 2-1. Create a VI

Your objective is to build a VI.

Imagine that you have sensors that read temperature and volume readings as voltage. You will use a VI in the `LabVIEW\Activity` directory to simulate the temperature and volume measurements in volts. You will write a VI to scale these measurements to degrees fahrenheit and liters, respectively.

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

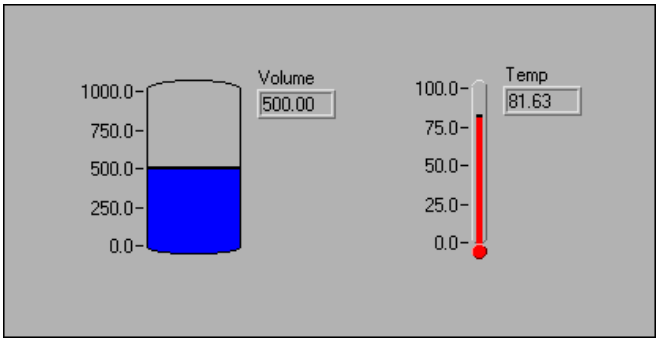
☞ **Note** ***If the* Controls *palette is not visible, select* Windows»Show Controls Palette *to display the palette. You also can access the* Controls *palette by popping up in an open area of the front panel. To pop up, right-click on your mouse (<Option>-click for Macintosh).***

2. Select **Tank** from **Controls»Numeric**, and place it on the front panel.
3. Type `Volume` in the label text box and click anywhere on the front panel.

☞ **Note** ***If you click outside the text box without entering text, the label disappears. To show the label again, pop up on the control and select* Show»Label.**

4. Rescale the tank indicator to display the tank volume between 0.0 and 1000.0.
   a. Using the Labeling tool, double-click on `10.0` on the tank scale to highlight it.
   b. Type `1000` in the scale and click the mouse button anywhere on the front panel. The intermediary increments are scaled automatically.
5. Place a thermometer from **Controls»Numeric** on the front panel. Label it `Temp` and rescale it to be between 0 and 100.

6. Your front panel should look like the following illustration.



7. Open the block diagram by choosing **Windows»Show Diagram**. Select the objects listed below from the **Functions** palette and place them on the block diagram.

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

8. Place each of the following objects on the block diagram.

Process Monitor (**Functions»Select a VI** from the `LabVIEW\Activity` directory)—Simulates reading a temperature voltage and volume value from a sensor or transducer.

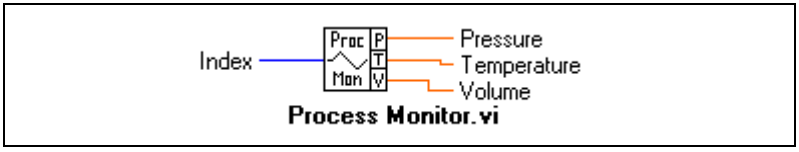
Random Number Generator (**Functions»Numeric**)—Generates a number between 0 and 1.

Multiply function (**Functions»Numeric**)—Multiplies two numbers and returns their product. In this activity, you need two of these. Drop one from the palette and copy and paste to create the other.

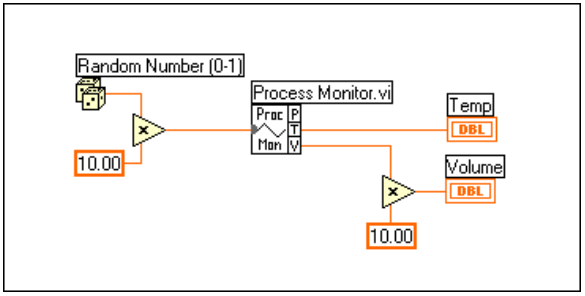
Numeric Constant (**Functions»Numeric**)—You need two of these. Drop one from the palette. Using the labeling tool, change its value to `10.00`. Copy and paste it.

Note ***Another way to create a constant is to pop up on the terminal of a function or VI using the Wiring tool. Select* Create Constant *from the floating menu. A constant of the appropriate data type appears.***

9. To view the inputs and outputs of a function or a VI, select **Show Help** from the **Help** menu and then drag the cursor over each function and VI. The Help window for the Process Monitor VI is shown below.

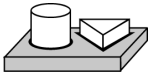


10. Using the Wiring tool, wire the objects as shown.



Note ***To move objects around on the block diagram, click on the Positioning tool in the* Tools *palette.***

11. Select **File»Save** and save the VI as `Temp & Vol.vi` in the `LabVIEW\Activity` directory.
12. From the front panel, run the VI by clicking on the **Run** button. Notice values for Volume and Temperature are displayed on the front panel.
13. Close the VI by selecting **File»Close**.



End of Activity 2-1.

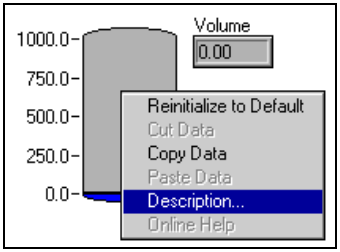
VI Documentation

You can document a VI by choosing **Windows»Show VI Info…**. Type the description of the VI in the VI Information dialog box. Then, you can recall the description by selecting **Windows»Show VI Info…** again.

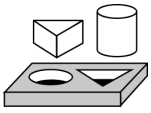
You can edit the descriptions of objects on the front panel (or their respective terminals on the block diagram) by popping up on the object and choosing **Data Operations»Description…**.

Note ***You cannot change the description of a VI or its front panel objects while the VI is running.***

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



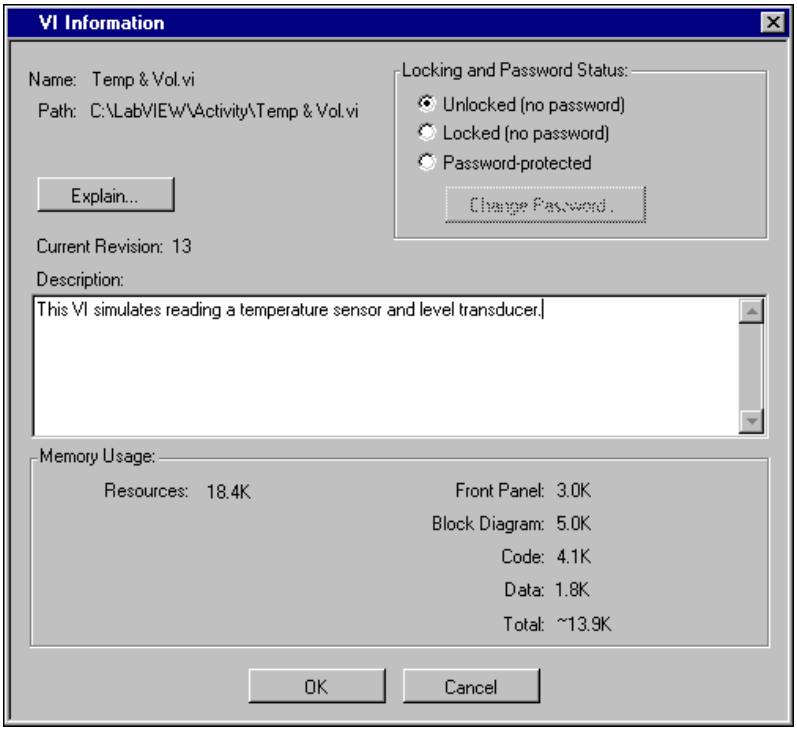
You also can view the description of a front panel object by showing the Help window (**Help»Show Help**) and moving the cursor over the object.



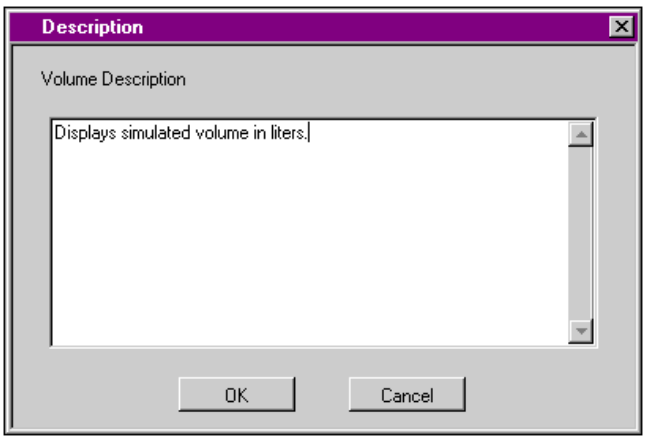
Activity 2-2. Document a VI

Your objective is to document a VI that you have created.

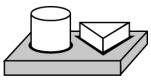
1. Open the `Temp & Vol.vi` created in Activity 2-1 from the `LabVIEW\Activity` directory.
2. Select **Windows»Show VI Info…**. Type the description for the VI, as shown in the following illustration, and click on **OK.**



3. Pop up on the tank and choose **Data Operations»Description…**. Type the description for the indicator, as shown in the following illustration, and click **OK**.



4. Pop up on the thermometer and choose **Data Operations» Description…**. Type in the description: `Displays simulated temperature (deg F) measurement.` Click on **OK**.
5. Select **Show Help** from the **Help** menu. Place the cursor on Volume and then on Temp. You can see the descriptions you typed in appear in the help window.
6. Save and close the VI.



End of Activity 2-2.

What is a SubVI?

A *subVI* is much like a subroutine in text-based programming languages. It is a VI that is used in the block diagram of another VI.

You can use any VI that has an icon and a connector as a subVI in another VI. In the block diagram, you select VIs to use as subVIs from **Functions»Select a VI…**. Choosing this option produces a file dialog box, from which you can select any VI in the system. If you open a VI that does not have an icon and a connector, a blank, square box appears in the calling VI's block diagram. You cannot wire to this node. For more information about icons and connectors, see the *LabVIEW Online Tutorial*, which you can access from the startup dialog box.

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

Hierarchy Window

The Hierarchy window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and global variables. You use the Hierarchy window (**Project»Show VI Hierarchy**) to display the dependencies of VIs by providing information on VI callers and subVIs. This window contains a toolbar that you can use to configure several types of settings for displayed items. The following illustration shows an example of the VI hierarchy toolbar.

You can use buttons on the Hierarchy window toolbar or the View menu, or pop up on an empty space in the window to access the following options. For more information about the Hierarchy window see the *Using the Hierarchy Window* section in Chapter 3, *Using SubVIs*, of the *G Programming Reference Manual*.

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

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

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

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

Include/Exclude global—Toggles the hierarchy graph to include or exclude global variables. Global variables store data used by several VIs.

Include/Exclude typedefs—Toggles the hierarchy graph to include or exclude typedefs. A typedef is a master copy of a custom control, which can be used by several VIs.

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

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

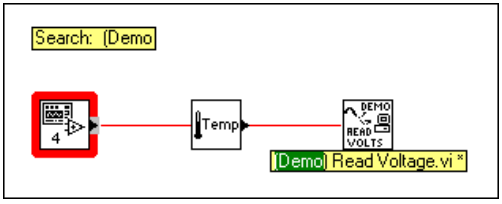
Use the <Tab> key to toggle between the Positioning and Scroll window tools. This feature is useful for moving nodes from the Hierarchy window to the block diagram.

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

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

Search Hierarchy

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

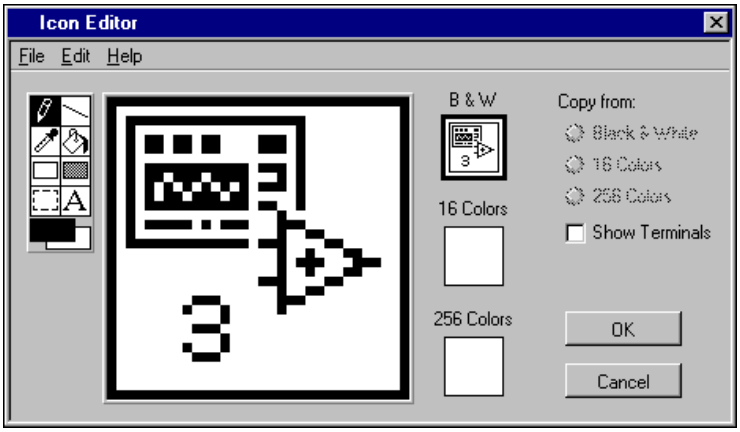


After finding the correct node, you can press <Enter> to search for the next node that matches the search string, or you can press <Shift-Enter> to find the previous node that matches the search string.

Icon and Connector

Every VI has a default icon displayed in the upper-right corner of the Front Panel and Diagram windows. For VIs, the default is the LabVIEW VI icon and a number indicating how many new VIs you have opened since launching LabVIEW. You use the Icon Editor to customize the icon by turning individual pixels on and off. To activate the Icon Editor, pop up on the default icon in the top right corner of the Panel window and select **Edit Icon**.

The following illustration shows the Icon Editor Window. You use the tools at left to create the icon design in the pixel editing area. An image of the actual icon size appears in one of the boxes to the right of the editing area.



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

Pencil tool—Draws and erases pixel by pixel.

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

Color Copy tool—Copies the foreground color from an element in the icon.

Fill bucket tool—Fills an outlined area with the foreground color.

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

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

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

Text tool—Enters text into the icon design.

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

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

- **Undo**—Cancels the last operation you performed.
- **OK**—Saves your drawing as the VI icon and returns to the front panel.
- **Cancel**—Returns to the front panel without saving any changes.

Depending on the type of monitor you are using, you can design a separate icon for monochrome, 16-color, and 256-color mode. You design and save each icon version separately. The editor defaults to **Black & White**, but you can click on one of the other color options to switch modes.

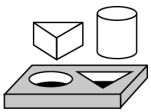
☞ **Note** ***If you design a color icon only, the icon does not show up in a subpalette of the* Functions *palette if you place the VI in the* `*.lib` *directory, nor will the icon be printed or displayed on a black and white monitor.***

The *connector* is the programmatic interface to a VI. If you use the panel controls or indicators to pass data to and from subVIs, these controls or indicators need terminals on the connector pane. You define connections by choosing the number of terminals you want for the VI and assigning a front panel control or indicator to each of those terminals.

To define a connector, select **Show Connector** from the icon pane pop-up menu on the Panel window.

The connector icon replaces the icon in the upper-right corner of the Panel window. LabVIEW selects a terminal pattern appropriate for your VI with terminals for controls on the left side of the connector pane, and terminals for indicators on the right. The number of terminals selected depends on the number of controls and indicators on your front panel.

Each rectangle on the connector represents a terminal area, and you can use the rectangles either for input or output from the VI. If necessary, you can select a different terminal pattern for your VI. To do this, pop up on the icon, select **Show Connector**, pop up again, and select **Patterns**.



Activity 2-3. Create an Icon and Connector

Your objective is to make an icon and connector for a VI.

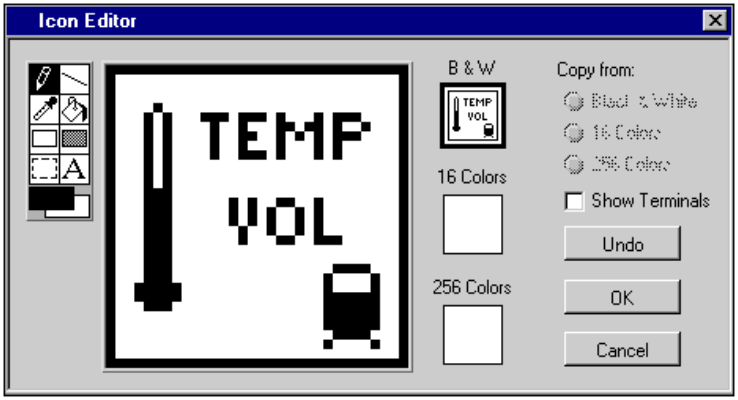
To use a VI as a subVI, you must create an icon to represent it on the block diagram of another VI, and a connector pane to which you can connect inputs and outputs. LabVIEW provides several tools with which you can create or edit an icon for your VIs.

The icon of a VI represents it as a subVI in the block diagram of other VIs. It can be a pictorial representation of the purpose of the VI, or a textual description of the VI.

1. Open `Temp & Vol.vi` in the `LabVIEW\Activity` directory.
2. From the front panel, pop up on the icon in the top right corner and select **Edit Icon…**. You also can double click on the icon to invoke the icon editor.

Note ***You only can access the icon/connector for a VI from the front panel.***

3. Erase the default icon. With the Select tool, which appears as a dotted rectangle, click and drag over the section you want to delete, and press the <Delete> key. You also can double click on the shaded rectangle in the tool box to erase the icon.
4. Draw a thermometer with the Pencil tool.
5. Create the text with the Text tool. To change the font, double-click on the Text tool. Your icon should look similar to the following illustration.

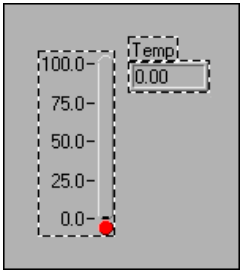


6. Close the Icon Editor by clicking on **OK**. The new icon appears in the icon pane.

7. Define the connector terminal pattern by popping up in the icon pane on the front panel and choosing **Show Connector**. By default, LabVIEW selects a terminal pattern based on the number of controls and indicators on the front panel. Because there are two objects on the front panel, the connector has two terminals, as shown at left.

8. Pop up on the connector pane and select **Rotate 90 Degrees**. Notice how the connector pane changes, as shown at left.
9. Assign the terminals to Temp and Volume.
   a. Click on the top terminal in the connector. The cursor automatically changes to the Wiring tool, and the terminal turns black.

   b. Click on the Temp indicator. A moving dashed line frames the indicator, as shown in the following illustration. The selected terminal changes to a color consistent with the datatype of the control/indicator selected.



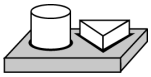
If you click in an open area on the front panel, the dashed line disappears and the selected terminal appears dimmed, indicating that you have assigned the indicator to that terminal. If the terminal is white, you have not made the connection correctly.

   c. Repeat steps a and b to associate the bottom terminal with the Volume indicator.
   d. Pop up on the connector and select **Show Icon…**.
10. Save the VI by choosing **File»Save**.

Now, this VI is complete and ready for use as a subVI in other VIs. The icon represents the VI in the block diagram of the calling VI. The connector (with two terminals) outputs the temperature and volume.

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

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

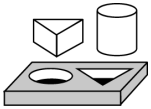


End of Activity 2-3.

Opening, Operating, and Changing SubVIs

You can open a VI used as a subVI from the block diagram of the calling VI by double-clicking on the subVI icon or by selecting **Project»This VI's SubVIs**. You will see a palette containing all the subVIs of the calling VI. Select the subVI you want to open.

Any changes you make to a subVI alter only the version in memory until you save the subVI. The changes affect all instances of the subVI and not just the node you used to edit the VI.



Activity 2-4. Call a SubVI

Your objective is to build a VI that uses the `Temp & Vol.vi` ***as a subVI.***

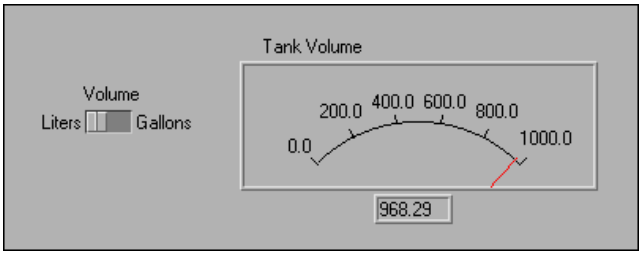
The Temp & Vol VI you built in Activity 2-1 returns a temperature and volume. You will take a volume reading and convert the value to gallons when a switch is pressed.

Front Panel

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

2. Select a Horizontal Switch from the **Controls»Boolean** palette and label it `volume`. Place free labels on the front panel to indicate `Liters` and `Gallons` by using the Labeling tool.

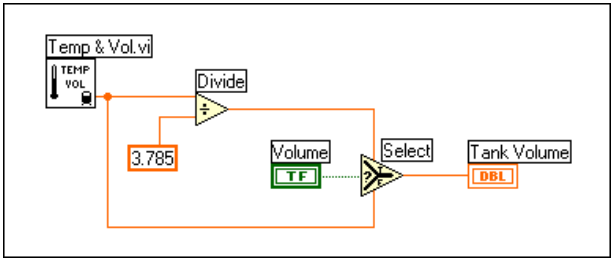
3. Select a meter from **Controls»Numeric** and place it on the front panel. Label it `Tank Volume`.



4. Change the range of the meter to accommodate values ranging between 0.0 and 1000.0. With the Operating tool, double-click on the high limit and change it from 10.0 to 1000.0. Switch to the positioning tool and resize the meter by dragging out one of the corners and expanding the control.

Block Diagram

5. Go to the block diagram by selecting **Windows»Show Diagram**.
6. Pop up in a free area of the block diagram and choose **Functions»Select a VI…**. A dialog box appears. Select `Temp & Vol.vi` in the `LabVIEW\Activity` directory. Click on **Open** in the dialog box. LabVIEW places the Temp & Vol VI on the block diagram.
7. Add the other objects to the block diagram as shown in the following illustration.



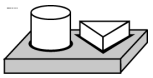
Numeric Constant (**Functions»Numeric**)—Add a numeric constant to the block diagram. Assign the value `3.785` to the constant by using the Labeling tool. This is the conversion factor for switching from liters to gallons.

Select Function (**Function»Comparison**)—Returns the value wired to the TRUE or FALSE input, depending on the Boolean input.

Divide function (**Functions»Numeric**)—Divides the value in liters by 3.785 to convert it to gallons.

8. Wire the diagram objects as shown.

9. Return to the front panel and click on the **Run** button in the toolbar. The meter shows the value in liters.
10. Click on the switch to select `Gallons` and click on the **Run** button. The meter shows the value in gallons.
11. Save the VI as `Using Temp & Vol.vi` in the `LabVIEW\Activity` directory.



End of Activity 2-4.

How Do You Debug a VI?

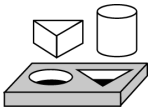
A VI cannot compile or run if it is broken. Normally, the VI is broken while you are creating or editing it, until you wire all the icons in the diagram. If it still is broken when you finish, try selecting **Remove Bad Wires** from the **Edit** menu. Often, this fixes a broken VI.

When your VI is not executable, a *broken* arrow appears instead of the **Run** button. To list the errors, click on the broken **Run** button. Click on one of the errors listed and then click on **Find** to highlight the object or terminal that reported the error.

You can animate the VI block diagram execution by clicking on the **Highlight Execution** button. Execution highlighting is commonly used with single-step mode to trace the data flow in a block diagram.

For debugging purposes, you might want to execute a block diagram node by node. This is known as single-stepping. To enable the single-step mode, click on the **Step Into** button or **Step Over** button. This action then causes the first node to blink, denoting that it is ready to execute. Then you can click on either the **Step Into** or **Step Over** button again to execute the node and proceed to the next node. If the node is a structure or VI, you can select the **Step Over** button to *execute* the node but *not single-step* through the node. For example, if the node is a subVI and you click on the **Step Over** button, you execute the subVI and proceed to the next node but cannot see how the subVI nodes execute. To single step through a structure or subVI, select the **Step Into** button.

Click on the **Step Out** button to finish execution of the block diagram nodes and/or complete single stepping. For more information about debugging, see Chapter 4, *Executing and Debugging VIs and SubVIs*, in the *G Programming Reference Manual*.

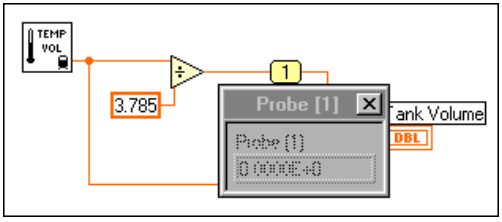


Activity 2-5. Debug a VI in LabVIEW

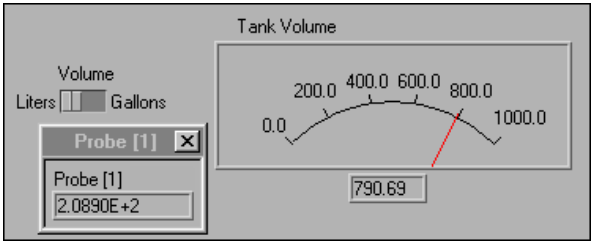
Your objective is to use the probe tool and the probe window and to examine data flow in the block diagram using the execution highlighting feature.

1. Open `Using Temp & Vol.vi` from the `LabVIEW\Activity` directory.
2. Select **Windows»Show Diagram**.
3. If the Tools palette is not open, select **Windows»Show Tools Palette**.

4. Select the Probe tool from the **Tools** palette. Click with the Probe tool on the wire coming out of the Divide function. A Probe window pops up with the title `Probe 1` and a yellow glyph with the number of the probe, as shown in the following illustration. The Probe window remains open, even if you switch to the front panel.



5. Return to the front panel. Move the Probe window so you can view both the probe and volume values as shown in the following illustration. Run the VI. The volume in gallons appears in the Probe window while `Tank Volume` displays the value in liters.



Note ***The volume values that appear on your screen may be different than what is shown in this illustration. Refer to the Numeric Conversion section in Chapter 3, Loops and Charts, for more information.***

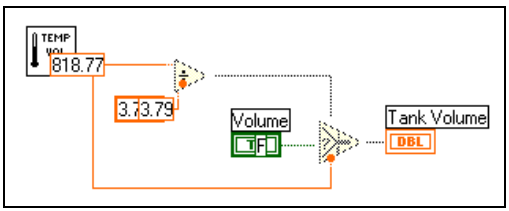
6. Close the Probe window by clicking in the close box at the top of the Probe window title bar.

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

7. Return to the block diagram of the VI.

8. Begin execution highlighting by clicking on the **Highlight Execution** button, in the toolbar. The **Highlight Execution** button changes to an illuminated light bulb.

9. Click on the **Run** button to run the VI, and notice that execution highlighting animates the VI block diagram execution. Moving bubbles represent the flow of data through the VI. Also notice that data values appear on the wires and display the values contained in the wires at that time, as shown in the following block diagram, just as if you had probed the wire.



You also can use the single stepping buttons if you want to walk through the graphical code, one step at a time.

10. Begin single-stepping by clicking on the **Step Over** button, in the toolbar.

11. Step into the Temp & Vol subVI by clicking on the **Step Into** button, in the toolbar. Clicking on this button opens the front panel and block diagram of your Temp & Vol subVI. Click on the **Step Over** button until the VI finishes executing.

12. Finish executing the block diagram by clicking on the **Step Out** button, in the toolbar. Clicking on this button completes all remaining sequences in the block diagram.

End of Activity 2-5.

3

Loops and Charts

This chapter introduces structures and explains the basic concepts of charts, the While Loop, and the For Loop. This chapter also provides activities that illustrate how to accomplish the following:

- Learn about different chart modes
- Use a While Loop and a chart
- Change the mechanical action of a Boolean switch
- Control loop timing
- Use a shift register
- Create a multiplot chart
- Use a For Loop

What is a Structure?

A *structure* is a program control element. Structures control the flow of data in a VI. G has five structures: the While Loop, the For Loop, the Case structure, the Sequence structure, and the Formula Node. This chapter introduces the While Loop and For Loop structures along with the chart and the shift register. The Case structure, Sequence structure, and Formula Node are explained in Chapter 4, *Case and Sequence Structures and the Formula Node*.

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

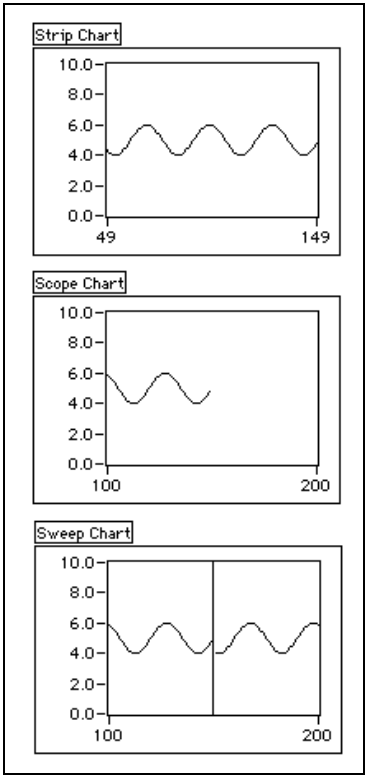
For examples of structures, see `Examples\General\structs.llb`. For examples of charts, see `Examples\General\Graphs\charts.llb`.

Charts

A *chart* is a numeric plotting indicator which is updated with new data periodically. You can find two types of charts in the **Controls»Graph** palette: waveform chart and intensity chart. You can customize charts to match your data display requirements or to display more information. Features available for charts include: a scrollbar, a legend, a palette, a digital display, and representation of scales with respect to time. For more information about charts, see Chapter 15, *Graph and Chart Controls and Indicators*, in your *G Programming Reference Manual*.

Chart Modes

The following illustration shows the three chart display options available from the **Data Operations»Update Mode** submenu—**Strip chart, Scope chart, and Sweep chart**. The default mode is strip chart.

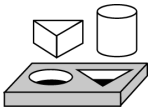


Faster Chart Updates

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

Overlaid Versus Stacked Plots

You can display multiple plots on a chart using a single vertical scale, called overlaid plots, or using multiple vertical scales, called stacked plots. Refer to the `charts.vi` example located in `Examples\General\Graphs\charts.llb`.



Activity 3-1. Experiment with Chart Modes

Your objective is to view a chart as your VI runs in strip chart mode, scope chart mode, and sweep chart mode.

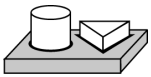
1. Open `Charts.vi`, located in the following directory: `LabVIEW\Examples\General\Graphs\charts.11b`.
2. Run the VI.

The strip chart mode has a scaling display similar to a paper tape strip chart recorder. As each new value is received, it is plotted at the right margin and old values shift to the left.

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

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

3. With the VI still running, pop up on any chart, and select **Update Mode**, and change the current mode to that of another chart. Notice the difference between the various charts and modes.
4. Stop and close the VI.

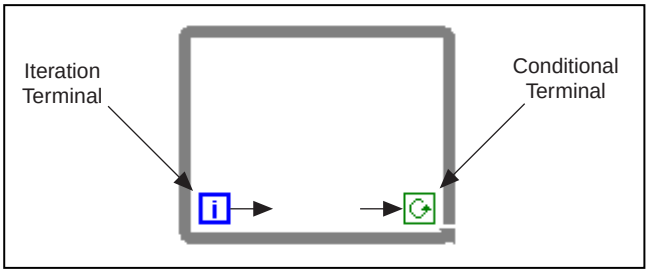


End of Activity 3-1.

While Loops

A While Loop is a structure that repeats a section of code until a condition is met. It is comparable to a Do Loop or a Repeat-Until Loop in traditional programming language.

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



The While Loop is equivalent to the following pseudocode:

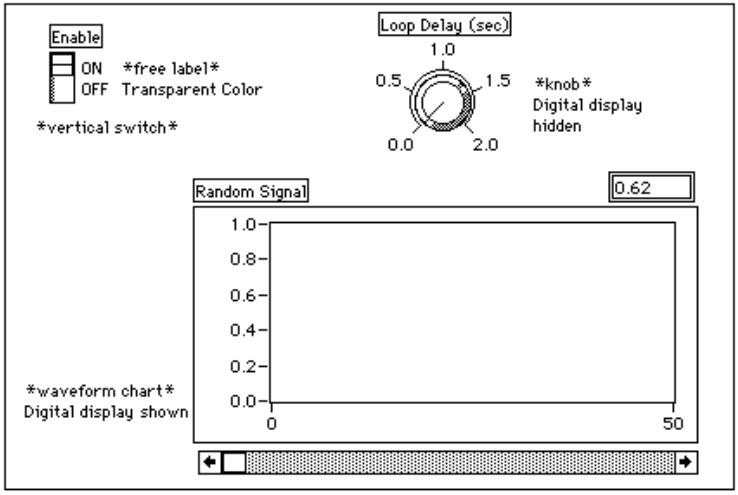
```
Do
Execute Diagram Inside the Loop (which sets the
condition)
While Condition is TRUE
```

Activity 3-2. Use a While Loop and a Chart

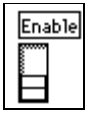
Your objective is to use a While Loop and a chart for acquiring and displaying data in real time.

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

Front Panel



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



2. Place a Vertical Switch (**Controls»Boolean**) on the front panel. Label the switch `Enable`.

3. Use the Labeling tool to create free labels for `ON` and `OFF`. Select the Labeling tool, and type in the label text. With the Color tool, shown at left, make the border of the free label transparent by selecting the T in the bottom left corner of the **Color** palette.

4. Place a waveform chart (**Controls»Graph**) on the front panel. Label the chart `Random Signal`. The chart displays random data in real time.

☞ **Note** ***Make sure that you select a waveform chart and not a waveform graph. In the* Graph *palette, the waveform chart appears closest to the left side.***

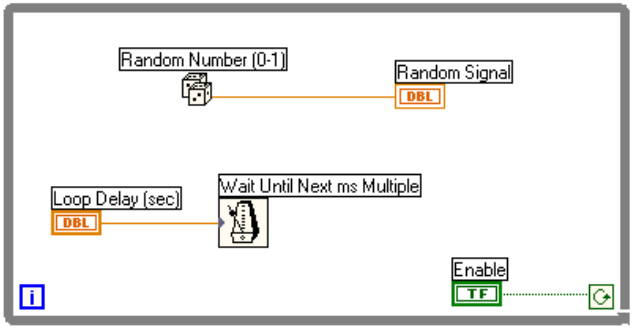
5. Pop up on the chart and choose **Show»Palette**, and **Show»Legend** to hide the palette and legend. The digital display shows the latest value. Then pop up on the chart and choose **Show»Digital Display** and **Show»Scroll Bar**.

6. Rescale the chart from `0.0` to `1.0`. Use the Labeling tool to replace the high limit of `10.0` with `1.0`.

7. Place a knob (**Controls»Numeric**) on the front panel. Label the knob `Loop Delay (sec)`. This knob controls the timing of the While Loop. Pop up on the knob and deselect **Show»Digital Display** to hide the digital display.
8. Rescale the knob. Using the Labeling tool, double-click on `10.0` in the scale around the knob, and replace it with `2.0`.

Block Diagram

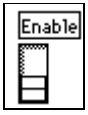
9. Open the block diagram and create the diagram in the following illustration.



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

b. Select the Random Number (0–1) function from **Functions» Numeric**.

c. Wire the diagram as shown in the Block Diagram, connecting the Random Number (0–1) function to the Random Signal chart terminal, and the Enable switch to the conditional terminal of the While Loop. Leave the Loop Delay terminal unwired for now.

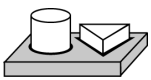


10. Return to the front panel and turn on the vertical switch by clicking on it with the Operating tool.
11. Save the VI as `Random Signal.vi` in the `LabVIEW\Activity` directory.
12. Run the VI.

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

13. Stop the VI by clicking on the vertical switch. Turning the switch off sends the value FALSE to the loop conditional terminal and stops the loop.
14. Scroll through the chart. Click and hold down the mouse button on either arrow in the scrollbar.
15. Clear the display buffer and reset the chart by popping up on the chart and choosing **Data Operations»Clear Chart**.

Note ***The display buffer default size is 1,024 points. You can increase or decrease this buffer size by popping up on the chart and choosing* Chart History Length…. *You only can use this feature when the VI is not running.***



End of Activity 3-2.

Mechanical Action of Boolean Switches

You might notice that each time you run the VI, you must turn on the vertical switch and then click the **Run** button in the toolbar. With G, you can modify the mechanical action of Boolean controls.

There are six possible choices for the mechanical action of a Boolean control:

- Switch When Pressed
- Switch When Released
- Switch Until Released
- Latch When Pressed
- Latch When Released
- Latch Until Released

Below are figures depicting each of these boolean switches, as well as a description of each of these mechanical actions.

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

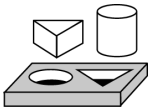
Switch When Released action—Changes the control value only after you release the mouse button, during a mouse click, within the graphical boundary of the control. The action is not affected by how often the VI reads the control. This action is similar to what happens when you click on a check mark in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

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

Latch When Pressed action—Changes the control value when you click on the control. It retains the new value until the VI reads it once, at which point the control reverts to its default value. (This action happens regardless of whether you continue to press the mouse button.) This action is similar to that of a circuit breaker and is useful for stopping While Loops or having the VI do something only once each time you set the control.

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

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

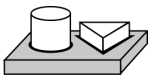


Activity 3-3. Change the Mechanical Action of a Boolean Switch

Your objective is to experiment with the different mechanical actions of Boolean switches.

1. Open the `Random Signal.vi`, as saved in Activity 3-2, from the `LabVIEW\Activity` directory. The default value of the `Enable` switch is FALSE.
2. Modify the vertical switch so it is used only to stop the VI. Change the switch so that you do not need to turn on the switch each time you run the VI.
   a. Turn on the vertical switch with the Operating tool.
   b. Pop up on the switch and choose **Data Operations»Make Current Value Default**. This makes the ON position the default value.
   c. Pop up on the switch and choose **Mechanical Action»Latch When Pressed**.
3. Run the VI. Click on the `Enable` switch to stop the acquisition. The switch moves to the OFF position momentarily and is reset back to the ON position.
4. Save the VI.

Note ***For your reference, LabVIEW contains an example that demonstrates these behaviors, called*** `Mechanical Action of Booleans.vi`***. It is located in*** `Examples\General\Controls\booleans.llb`***.***



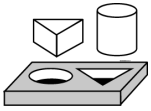
End of Activity 3-3.

Timing

When you ran the VI in the previous activity, the While Loop executed as quickly as possible. However, you can slow it down to iterate at certain intervals with the functions in the **Functions»Time & Dialog** palette.

The timing functions express time in milliseconds (ms), however, your operating system might not maintain this level of timing accuracy.

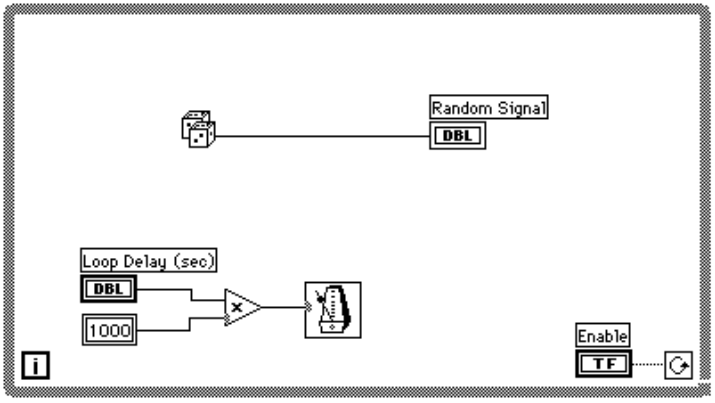
- **(Windows 95/NT)** The timer has a resolution of 1 ms. However, this is hardware-dependent, so on slower systems, such as an 80386, you might have lower resolution timing.
- **(Windows 3.1)** The timer has a default resolution of 55 ms. You can configure LabVIEW to have 1 ms resolution by selecting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



Activity 3-4. Control Loop Timing

Your objective is to control loop timing and ensure that no iteration is shorter than the specified number of milliseconds.

1. Open `Random Signal.vi`, as modified and saved in Activity 3-3, from the `LabVIEW\Activity` directory.
2. Modify the VI to generate a new random number at a time interval specified by the knob, as shown in the following illustration.

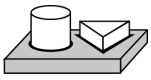


Wait Until Next ms Multiple function (**Functions»Time & Dialog**)—Multiply the knob terminal by 1,000 to convert the knob value in seconds to milliseconds. Use this value as the input to the Wait Until Next ms Multiple function.

Multiply function (**Functions»Numeric**)—The multiply function multiplies the knob value by 1000 to convert seconds to milliseconds.

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

3. Run the VI. Rotate the knob to get different values for the loop delay. Notice the effects of the loop delay on the update of the `Random Signal` display.
4. Save the VI as `Random Signal with Delay.vi` in the `LabVIEW\Activity` directory. Close the VI.

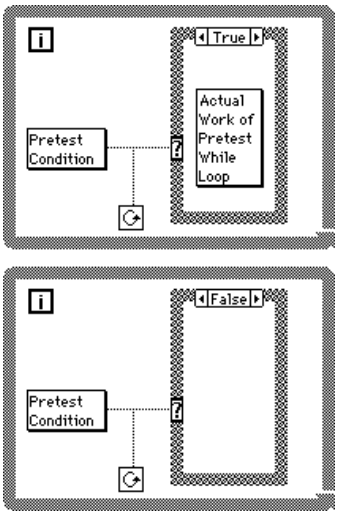


End of Activity 3-4.

Preventing Code Execution in the First Iteration

The While Loop always executes at least once, because G performs the loop test for continuation after the diagram executes. You can construct a While Loop that pretests its conditional terminal by including a Case structure inside the loop. Wire a Boolean input to the Case structure selector terminal so the subdiagram for the FALSE condition executes if the code in the While Loop should not execute. See Chapter 4, *Case and Sequence Structures and the Formula Node* for more information about using Case structures.

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

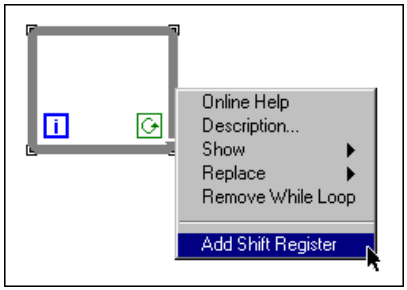


This example has the same result as the following pseudocode:

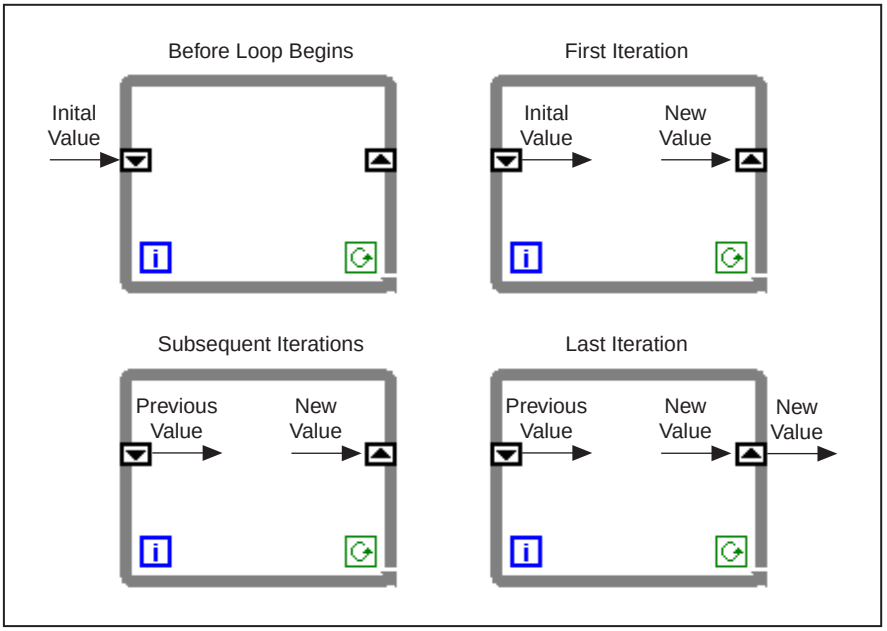
```
While (pretest condition)
Do actual work of While Loop
Loop
```

Shift Registers

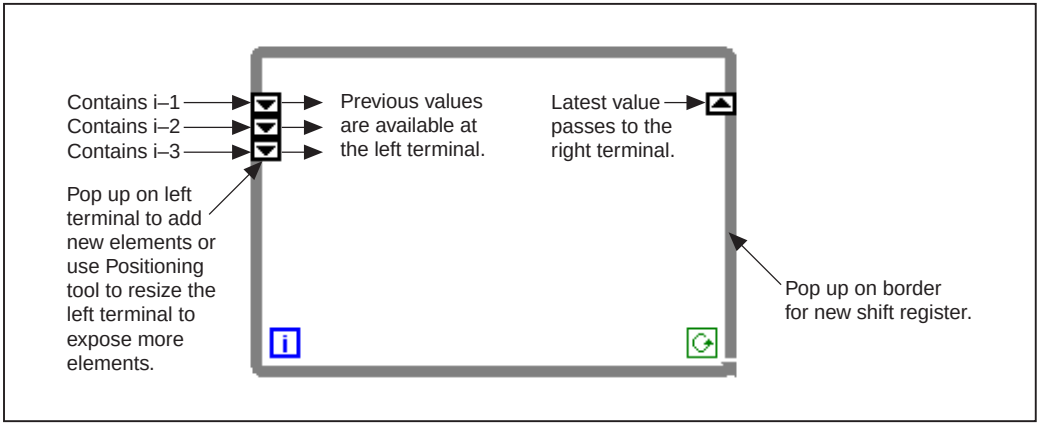
Shift registers (available for While Loops and For Loops) transfer values from one loop iteration to the next. You can create a shift register by popping up on the left or right border of a loop and selecting **Add Shift Register**.



The shift register contains a pair of terminals directly opposite each other on the vertical sides of the loop border. The right terminal stores the data upon the completion of an iteration. That data shifts at the end of the iteration and appears in the left terminal at the beginning of the next iteration, as shown in the following illustration. A shift register can hold any data type—numeric, Boolean, string, array, and so on. The shift register automatically adapts to the data type of the first object you wire to the shift register.



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

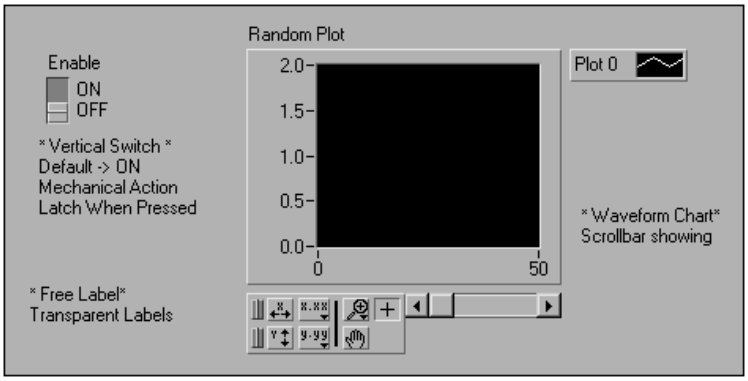


Activity 3-5. Use a Shift Register

Your objective is to build a VI that displays a running average on a chart.

Front Panel

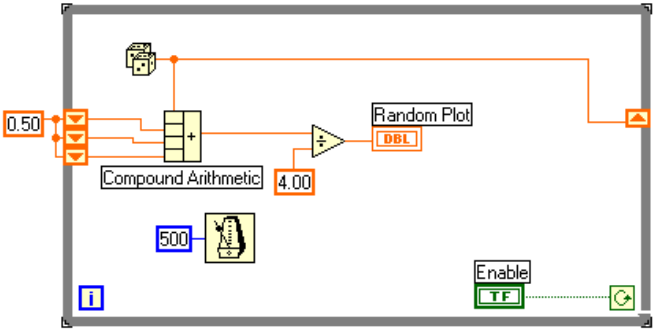
1. Open a new front panel and create the objects as shown in the following illustration.



2. Change the scale of the Waveform chart to range from `0.0` to `2.0`.
3. After adding the vertical switch, pop up on it and select **Mechanical Action»Latch When Pressed** and set the ON state to be the default by choosing **Operate»Make Current Values Default**.

Block Diagram

4. Build the block diagram shown in the following illustration.



5. Add the While Loop (**Functions»Structures**) in the block diagram and create the shift register.

   a. Pop up on the left or right border of the While Loop and choose **Add Shift Register**.

   b. Add an extra element by popping up on the left terminal of the shift register and choosing **Add Element**. Add a third element in the same manner as the second.

Random Number (0–1) function (**Functions»Numeric**)—This function generates random data ranging between 0 and 1.

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

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

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

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

6. Pop up on the input of the Wait Until Next ms Multiple function and select **Create Constant**. A numeric constant appears and is automatically wired to the function.

7. Type `500` in the label. The numeric constant wired to the Wait Until Next ms Multiple function specifies a wait of 500 milliseconds (one half-second). Thus, the loop executes once every half-second.

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

8. Run the VI and observe the operation.
9. Save this VI as `Random Average.vi` in the `LabVIEW\Activity` directory.

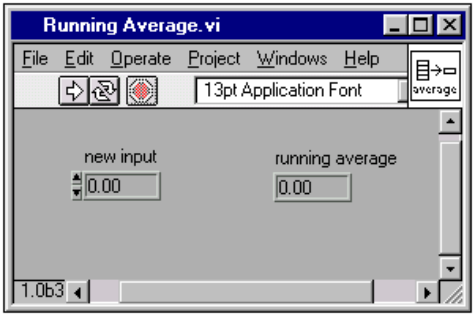
Note ***Remember to initialize shift registers to avoid incorporating old or default data into your current data measurements***

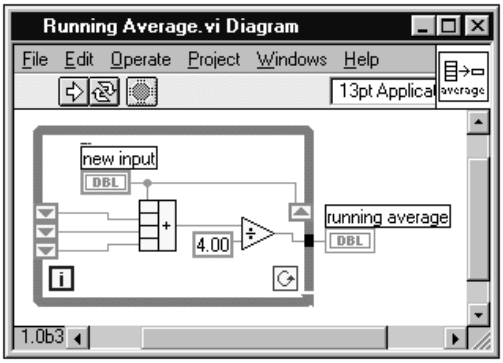
End of Activity 3-5.

Using Uninitialized Shift Registers

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

The following illustration shows an example of a subVI that calculates the running average of four data points. The VI uses an uninitialized shift register (with three additional elements) to store previous data points.





Each time the VI is called, `running average` is computed from the new input and the previous three values. Then the new value is saved into the shift register, and the previous two values are moved up in the shift register. There is no input value wired to the input side of the left shift registers, so all three values are preserved for the next execution of the VI.

Because this subVI has nothing wired to the condition terminal, it executes exactly once when called. The While Loop in this subVI is not used to loop several times, but to store values in the loop shift registers between calls.

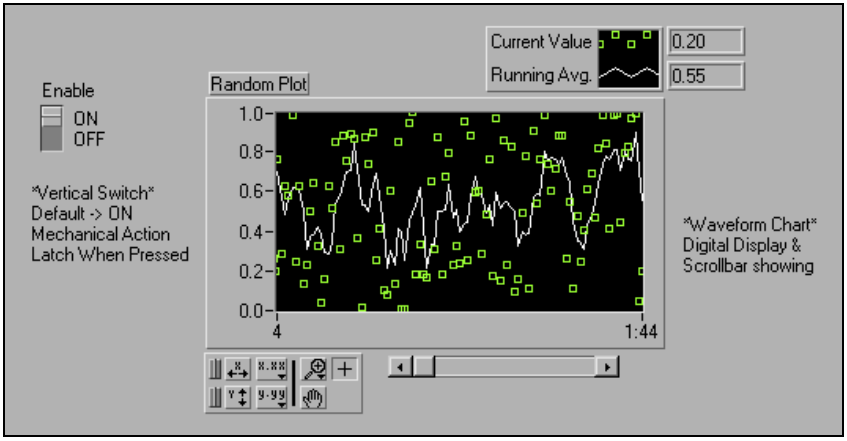
When the Running Average VI is loaded into memory, the uninitialized shift registers are set to zero automatically. If the shift registers are wired to Boolean values, the initial value is FALSE.

Activity 3-6. Create a Multiplot Chart

Your objective is to create a chart that can accommodate more than one plot.

Front Panel

1. Open the `Random Average.vi` you created in Activity 3-5.
2. Modify the Front Panel as shown in the following illustration.



a. Using the Positioning tool, stretch the legend to include two plots.

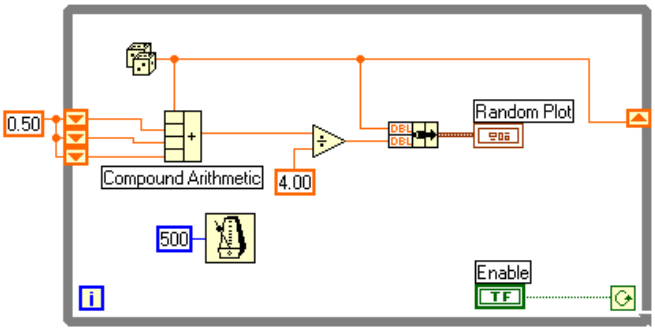
b. Show the digital display by popping up on the chart, and choosing **Show»Digital Display**. Move the legend if necessary.

c. Rename Plot 0 to `Current Value` by double-clicking on the label with the Labeling tool and typing in the new text. You can resize the label area by dragging either of the left corners with the Positioning tool. Rename Plot 1 to `Running Avg` in the same way.

d. For the `Current Value` plot, change the interpolation to unconnected, the point style to square, and the color to green. You can change the plot style and color by popping up on the legend.

Block Diagram

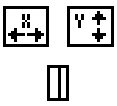
3. Modify the block diagram, as shown in the following illustration, to display both the average and the current random number on the same chart.



Bundle function (**Functions»Cluster**)—In this activity, the Bundle function bundles the average and current value for plotting on the chart. The bundle node appears as shown at left when you place it in the block diagram. You can add additional elements by using the Resizing cursor (accessed by placing the Positioning tool at the corner of the function) to enlarge the node.

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

4. From the front panel, run the VI. The VI displays two plots on the chart. The plots are overlaid. That is, they share the same vertical scale.
5. From the block diagram, run the VI with execution highlighting turned on to see the data in the shift registers.
6. Turn execution highlighting off. From the front panel, run the VI. While the VI is running, use the buttons from the palette to modify the chart. You can reset the chart, scale the X or Y axis, and change the display format at any time. You also can scroll to view other areas or zoom into areas of a graph or chart.

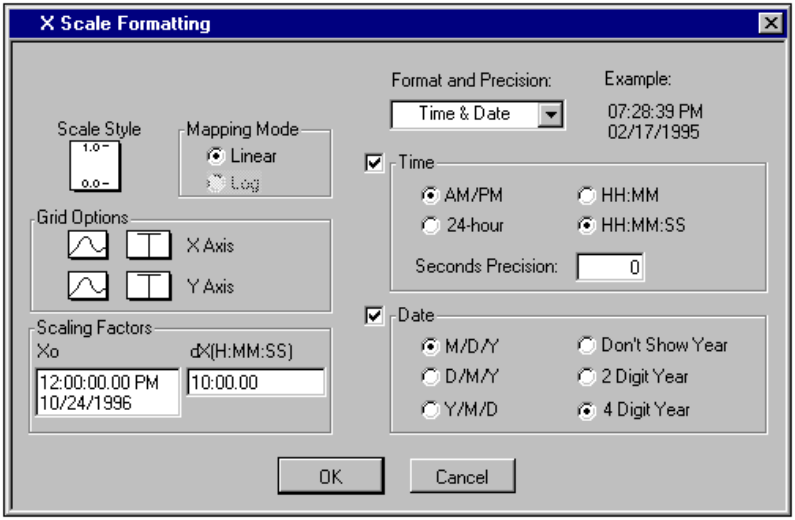


You can use the **X** and **Y** buttons to rescale the X and Y axes, respectively. If you want the graph to autoscale either of the scales continuously, click on the lock switch to the left of each button to lock on autoscaling.

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

7. Format the scales of the waveform chart to represent either absolute or relative time. To select the x scale time format, pop up on the x-scale and select **Formatting…**.

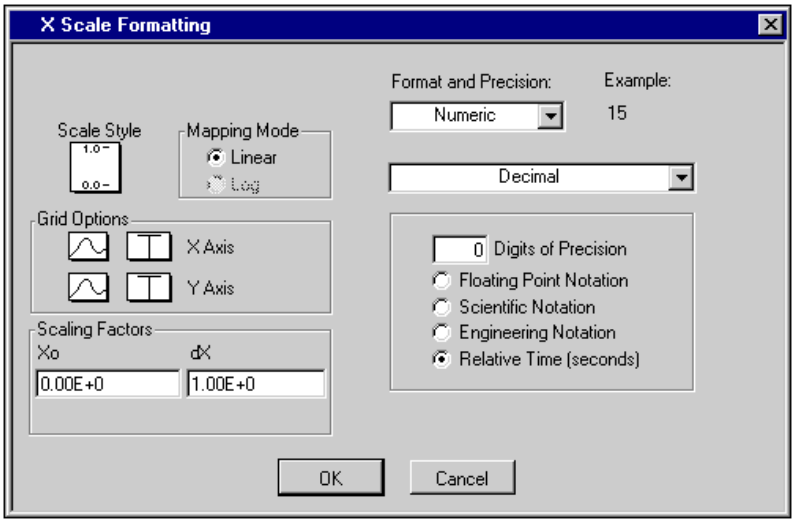
   a. Choose absolute time by selecting the **Time & Date** option from the **Format and Precision** menu ring. This changes the dialog box to the one shown below. For the waveform chart to start at a certain time and increment at certain intervals, you can edit the *Xo* and *dX* values respectively.



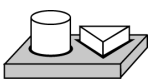
   b. Format the chart to display the data starting from noon, Oct. 24, 1996 and increment every 10 minutes, as shown above.

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

8. To select the relative time format, select **Numeric** from the **Format and Precision** menu ring. Then you can select the **Relative Time (seconds)** option in the dialog box and represent the time in seconds. Modify the dialog box, as shown in the following illustration, and select **OK**.



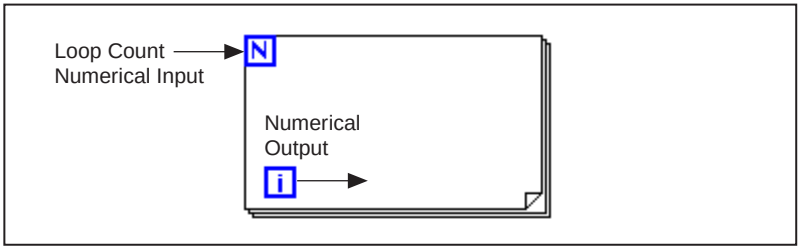
9. Run the VI.
10. Save the VI as `Multiple Random Plot.vi` in the `LabVIEW\Activity` directory.



End of Activity 3-6.

For Loops

A For Loop executes a section of code a defined number of times. It is resizable, and, like the While Loop, is not dropped on the block diagram immediately. Instead, a small icon representing the For Loop appears in the block diagram, and you have the opportunity to size and position it. To do so, first click in an area above and to the left of all the terminals. While holding down the mouse button, drag out a rectangle that encompasses the terminals you want to place inside the For Loop. When you release the mouse button, G creates a For Loop of the size and position you selected. You place the For Loop on the block diagram by selecting it from **Functions»Structures**.



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

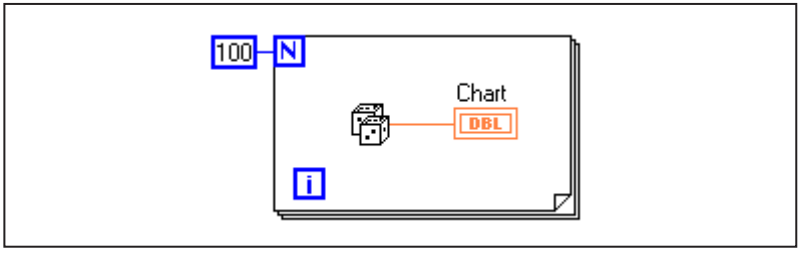
N Count terminal (an input terminal)—The count terminal specifies the number of times to execute the loop.

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

The For Loop is equivalent to the following pseudocode:

```
For i = 0 to N-1
Execute Diagram Inside The Loop
```

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

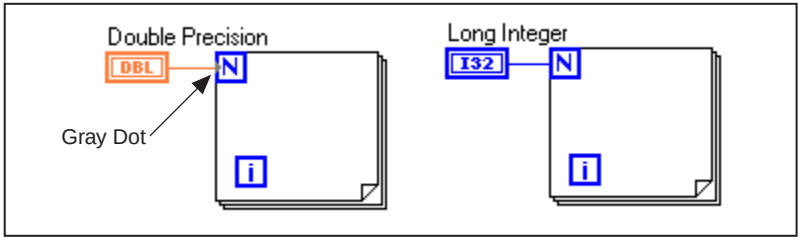


Numeric Conversion

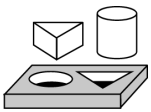
Until now, all the numeric controls and indicators you have used have been double-precision, floating-point numbers represented with 32 bits. G, however, can represent numerics as integers (byte, word, or long) or floating-point numbers (single-, double-, or extended-precision). The default representation for a numeric is a double-precision, floating-point.

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

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



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

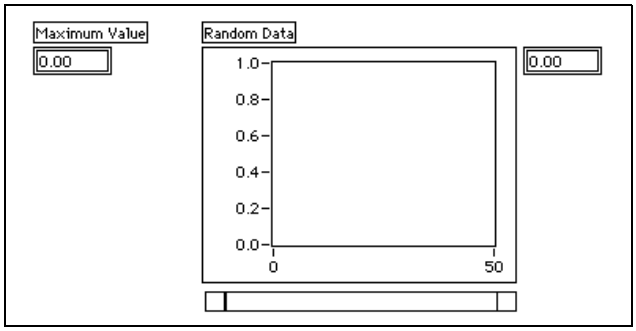


Activity 3-7. Use a For Loop

Your objective is to use a For Loop and shift registers to calculate the maximum value in a series of random numbers.

Front Panel

1. Open a new front panel and add the objects shown in the following illustration.



a. Place a digital indicator on the front panel and label it `Maximum Value`.

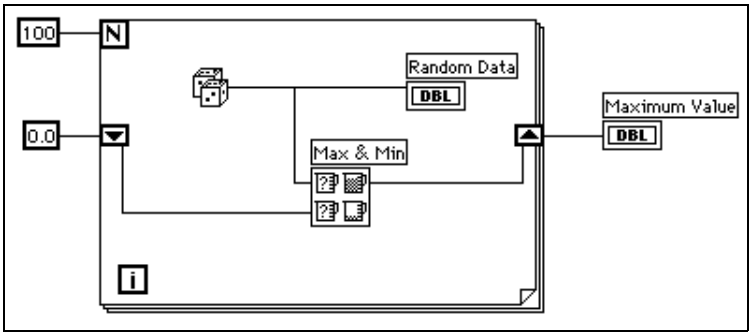
b. Place a waveform chart on the front panel and label it `Random Data`. Change the scale of the chart to range from `0.0` to `1.0`.

c. Pop up on the chart and choose **Show»Scrollbar** and **Show»Digital Display**. Pop up and hide the palette and legend.

d. Resize the scrollbar with the positioning tool.

Block Diagram

2. Open the block diagram and modify it as shown in the following illustration.



3. Place a For Loop (**Functions»Structures**) on the block diagram.
4. Add the shift register by popping up or right-clicking on the right or left border of the For Loop and choosing **Add Shift Register**.
5. Add the following objects to the block diagram.

Random Number (0–1) function (**Functions»Numeric**)—This function generates the random data.

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

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

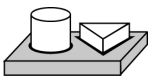
You must know something about the data you are collecting to initialize a shift register. For example, if you initialize the shift register to 1.0, then that value is already greater than all the expected data values, and is always the maximum value. If you did not initialize the shift register, then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

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

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

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

7. Run the VI.
8. Save the VI as `Calculate Max.vi` in the `LabVIEW\Activity` directory.



End of Activity 3-7.

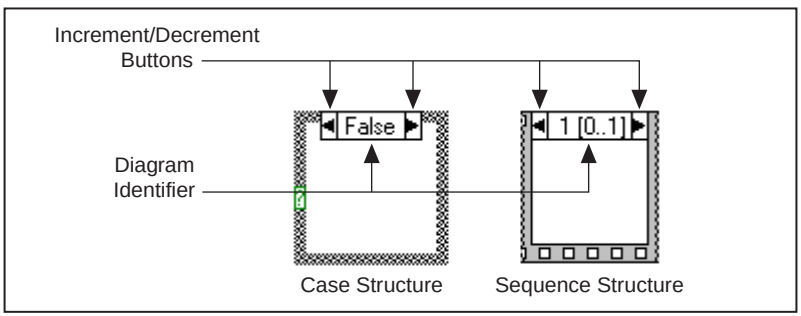
4

Case and Sequence Structures and the Formula Node

This chapter introduces the basic concepts of Case and Sequence structures and the Formula Node, and provides activities that explain the following:

- How to use the Case structure
- How to use the Sequence structure
- What sequence locals are and how to use them
- What a Formula Node is and how to use it

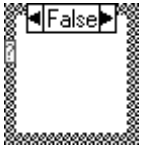
Both Case and Sequence structures can have multiple subdiagrams, configured like a deck of cards, of which only one is visible at a time. At the top of each structure border is the *subdiagram display window*, which contains a *diagram identifier* in the center and decrement and increment buttons at each side. The diagram identifier indicates which subdiagram currently is displayed. For Case structures, a diagram identifier is a list of values which select the subdiagram. For Sequence structures, a diagram identifier is the number of the frame in the sequence (0 to *n* – 1). The following illustration shows a Case structure and a Sequence structure.



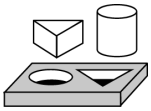
Clicking on the decrement (left) or increment (right) button displays the previous or next subdiagram, respectively. Incrementing from the last subdiagram displays the first subdiagram, and decrementing from the first subdiagram displays the last. For more information about Case and Sequence structures, refer to Chapter 19, *Structures*, in the *G Programming Reference Manual*.

Case Structure

The Case structure has two or more subdiagrams, or *cases*, exactly one of which executes when the structure executes. This depends on the value of an integer, Boolean, string, or enum value you wire to the external side of the selection terminal or *selector*. A Case structure is shown in the following illustration.



Note ***Case statements in other programming languages generally do not execute any case if a case is out of range. In G, you must either include a default case that handles out-of-range values or explicitly list every possible input value.***

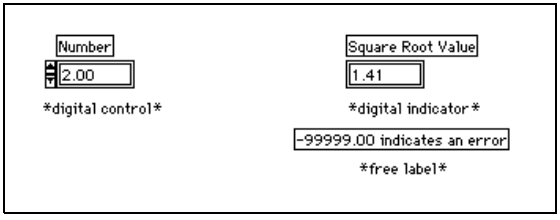


Activity 4-1. Use the Case Structure

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

Front Panel

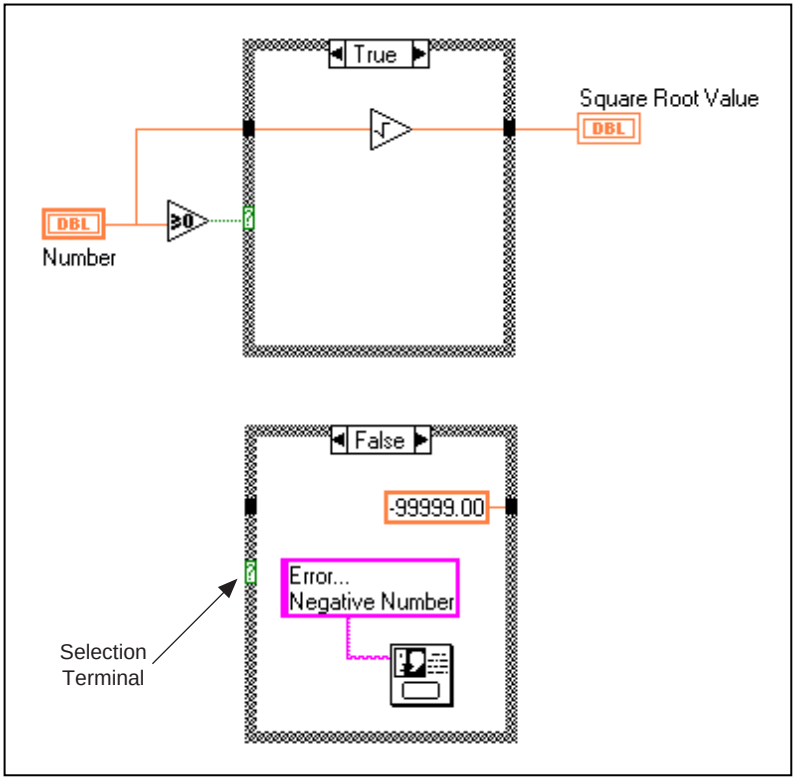
1. Open a new front panel and create the objects as shown in the following illustration.



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

Block Diagram

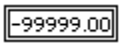
2. Build the diagram as shown in the following illustration.



3. Place a Case structure in the block diagram by selecting it from **Functions»Structures**. The Case structure is a resizable box that is not dropped on the diagram immediately. Instead, you have the chance to position it and resize it. To do so, click in an area above and to the left of all the terminals you want to be inside the Case structure. Continue holding down the mouse button and drag out a rectangle that encompasses the terminals.

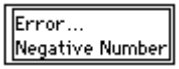
Greater Or Equal To 0? function (**Functions»Comparison**)—Returns a TRUE if the number input is greater than or equal to 0.

Square Root function (**Functions»Numeric**)—Returns the square root of the input number.



Numeric Constant (**Functions»Numeric**)—In this activity, the constant indicates the numeric value of the error.

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



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

The VI executes either the TRUE case or the FALSE case. If the number is greater than or equal to zero, the VI executes the TRUE case and returns the square root of the number. The FALSE case outputs –99999.00 and displays a dialog box with the message `Error...Negative Number`.

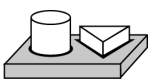
Note ***You must define the output tunnel for each case. When you create an output tunnel in one case, tunnels appear at the same position in all the other cases. Unwired tunnels appear as white squares.***

4. Return to the front panel and run the VI. Try a number greater than zero and a number less than zero by changing the value in the digital control you labeled `Number`. Notice that when you change the digital control to a negative number, LabVIEW displays the error message you set up in the FALSE case of the Case structure.
5. Save the VI as `Square Root.vi` in the `LabVIEW\Activity` directory.

VI Logic

The block diagram in this activity has the same effect as the following pseudocode in a text-based language.

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

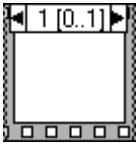


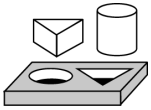
End of Activity 4-1.

Sequence Structures

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

A Sequence structure is shown in the following illustration.



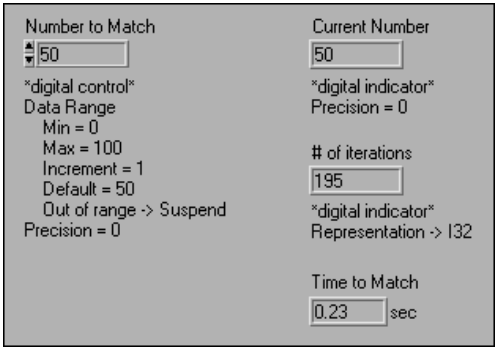


Activity 4-2. Use a Sequence Structure

Your objective is to build a VI that computes the time it takes to generate a random number that matches a given number.

Front Panel

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

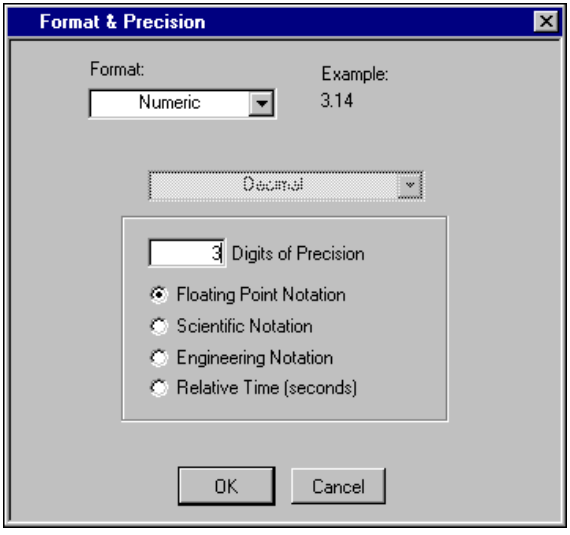


The Number to Match control contains the number you want to match. The Current Number indicator displays the current random number. The # of iterations indicator displays the number of iterations before a match. Time to Match indicates how many seconds it took to find the matching number.

Modifying the Numeric Format

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

2. Pop up on the Time to Match digital indicator and choose **Format & Precision…**. The front panel must be the active window to access the menu.
3. Enter 3 for Digits of Precision and click **OK**.

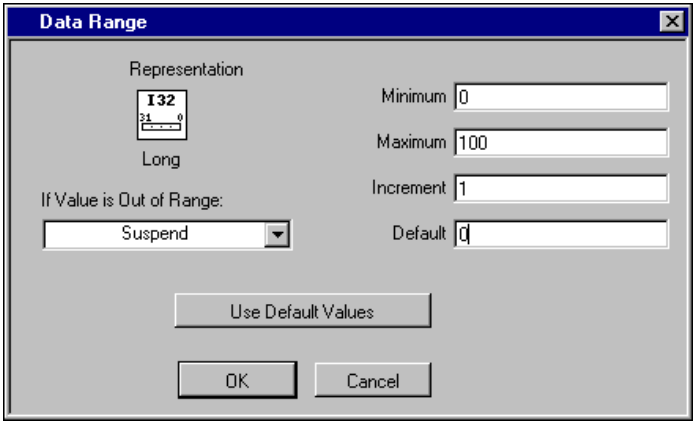


4. Pop up on the Number to Match digital control and choose **Representation»I32**.
5. Repeat Step 4 for the Current Number and the # of iterations digital indicators.

Setting the Data Range

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

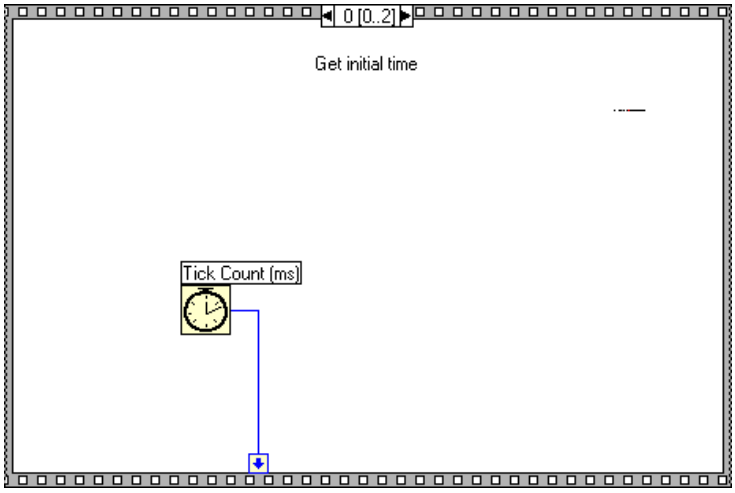
6. Pop up on the Number to Match indicator and choose **Data Range…**.
7. Fill in the dialog box as shown in the following illustration and click **OK**.

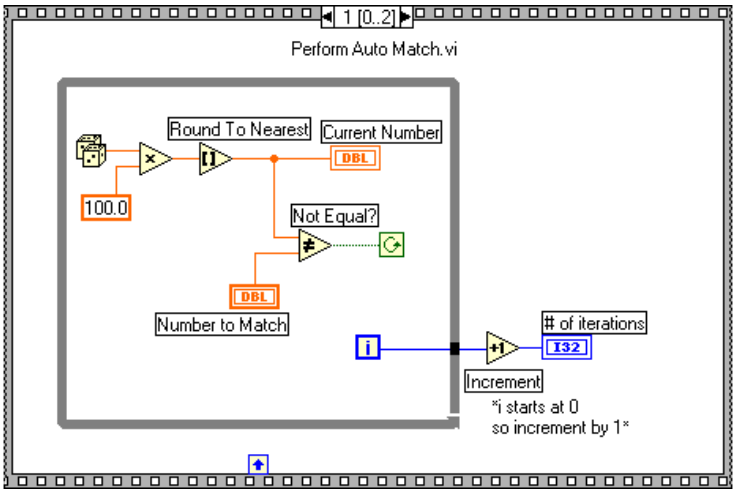


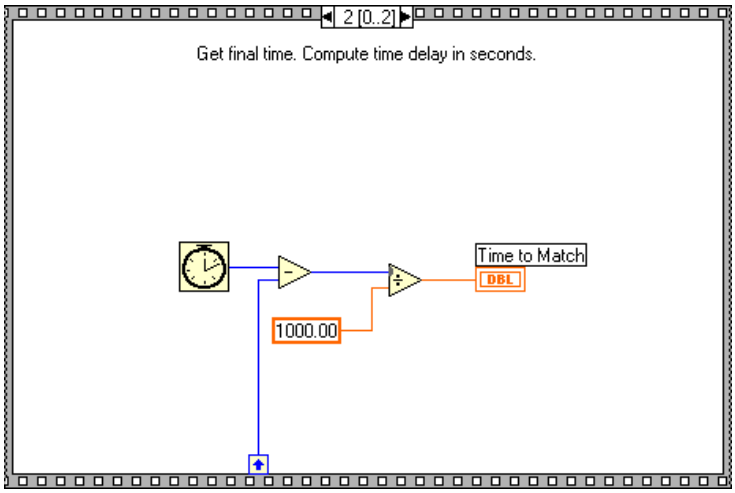
Block Diagram

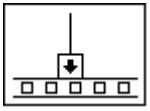
8. Open the block diagram.
9. Place the Sequence structure (**Functions»Structures**) in the block diagram.
10. Enlarge the structure by dragging one corner with the Resizing cursor.

11. Create a new frame by popping up on the frame border and choose **Add Frame After**. Repeat this step to create frame 2.
12. Build the block diagram shown in the following illustrations.









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

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

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

14. Finish the block diagram as shown in the opening illustration of the *Block Diagram* section in this activity.

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

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

Multiply function (**Functions»Numeric**)—In this activity, the function multiplies the random number by 100.

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

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

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

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

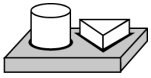
Subtract function (**Functions»Numeric**)—In this activity, the function returns the time (in milliseconds) elapsed between frame 2 and frame 0.

Divide function (**Functions»Numeric**)—In this activity, the function divides the number of milliseconds elapsed by 1,000 to convert the number to seconds.

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

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

15. Return to the front panel and enter a number inside the `Number to Match` control and run the VI.
16. Save the VI as `Time to Match.vi` in the `LabVIEW\Activity` directory.



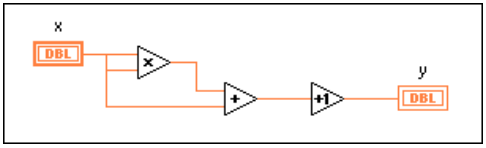
End of Activity 4-2.

Formula Node

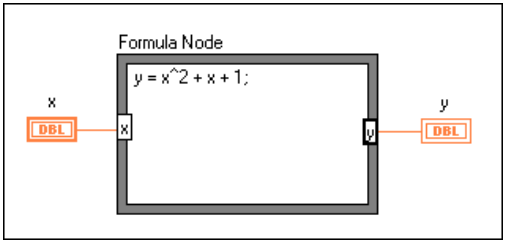
The Formula Node is a resizable box that you can use to enter formulas directly into a block diagram. You place the Formula Node on the block diagram by selecting it from **Functions»Structures**. This feature is useful when an equation has many variables or is otherwise complicated. For example, consider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

$y = x^2 + x + 1$

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

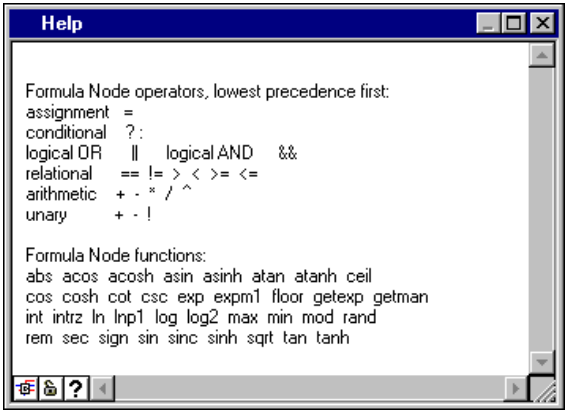


You can implement the same equation using a Formula Node, as shown in the following illustration



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

The operators and functions available inside the Formula Node are listed in the Help window for the Formula Node, as shown in the following illustration. A semicolon terminates each formula statement.

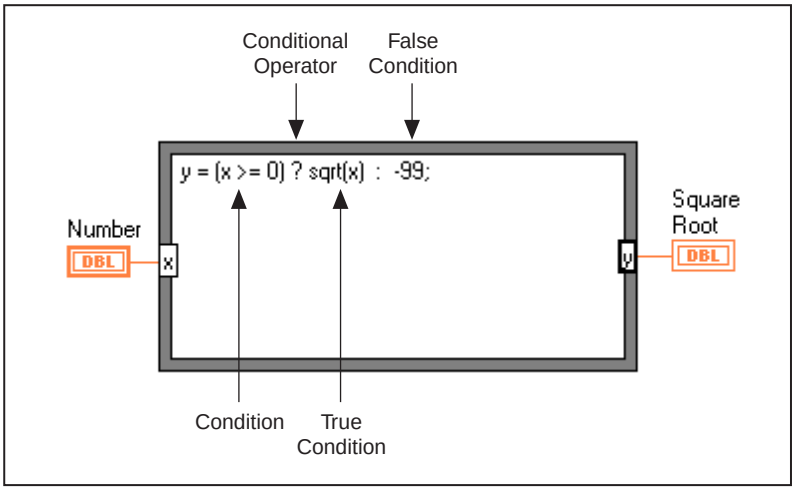


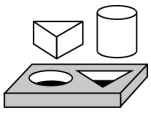
The following example shows how you can perform a conditional assignment inside a Formula Node.

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

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

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





Activity 4-3. Use the Formula Node

Your objective is to build a VI that uses the Formula Node to calculate the following equations.

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

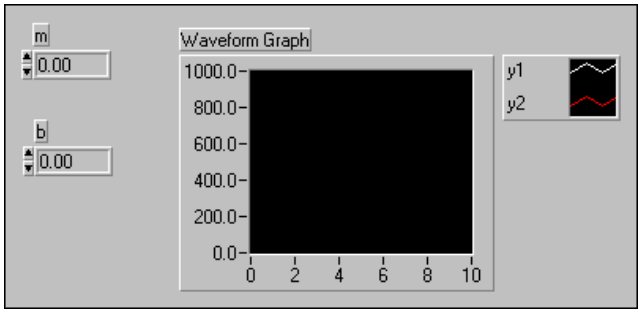
$y2 = m * x + b$

where x ranges from 0 to 10.

You will use only one Formula Node for both equations, and you will graph the results on the same graph. For more information on graphs, see Chapter 5, *Arrays, Clusters, and Graphs*.

Front Panel

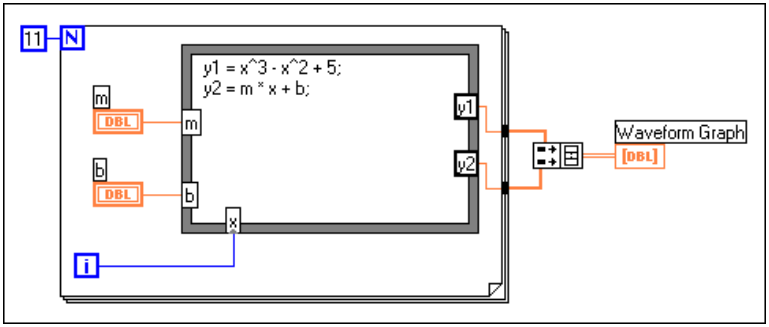
1. Open a new front panel and build the front panel shown in following illustration. The waveform graph indicator displays the plots of the equation. The VI uses the two digital controls to input the values for `m` and `b`.

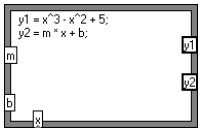


2. Create the graph legend shown in the following illustration by selecting **Show»Legend**. Use the Resizing cursor to drag the legend downward so it displays two plots. Use the Labeling tool to rename the plots. You can define the line style for each plot using the legend pop-up menu. You also can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

Block Diagram

3. Build the block diagram shown in the following illustration.





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

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

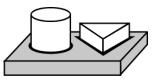
Note ***Although variable names are not limited in length, be aware that long names take up considerable diagram space. A semicolon (;) terminates the formula statement.***

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

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

Build Array (**Functions»Array**) puts two array inputs into the form of a multiplot graph. Create the two input terminals by using the Resizing cursor to drag one of the corners. For more information on arrays, see Chapter 5, *Arrays, Clusters, and Graphs*.

4. Return to the front panel and run the VI with different values for `m` and `b`.
5. Save the VI as `Equations.vi` in the `LabVIEW/Activity` directory.



End of Activity 4-3.

Artificial Data Dependency

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

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

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

5

Arrays, Clusters, and Graphs

This chapter introduces the basic concepts of polymorphism, arrays, clusters, and graphs and provides activities that explain auto-indexing and the Graph and Analysis VIs.

Arrays

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

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

How Do You Create and Initialize Arrays?

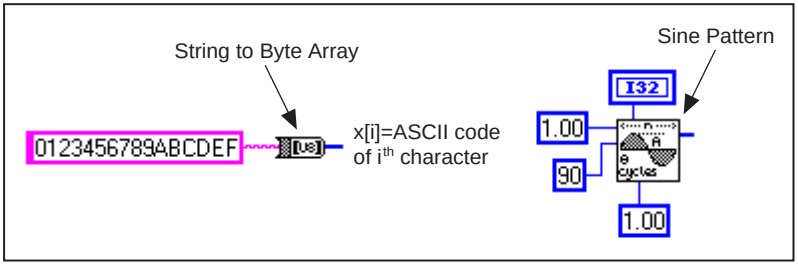
If you need an array as a source of data in your block diagram, you can choose **Functions»Array** and then select and place the array shell on your block diagram. Using the Operating tool, you can choose a numeric constant, Boolean constant, or string constant to place inside the empty array. The following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

To create an array on the front panel, select **Array & Cluster** from the **Controls** palette and place the array shell on your front panel. Then select an object (numeric, for example) and place that inside the array shell. This creates an array of numerics.

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

For more information on how to create array controls and indicators on the front panel, see Chapter 14, *Array and Cluster Controls and Indicators*, in the *G Programming Reference Manual*.

There are several ways to create and initialize arrays on the block diagram. Some block diagram functions also produce arrays, as the following illustration shows.



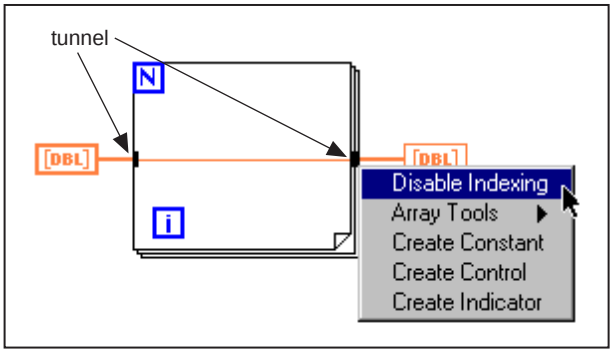
Array Controls, Constants, and Indicators

You create array controls, constants, and indicators on the front panel or block diagram by combining an array shell with a numeric, Boolean, string, or cluster. An array element cannot be another array, chart, or graph. For examples of arrays, see `Examples\General\arrays.llb`.

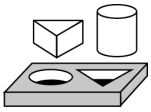
Auto-Indexing

For Loop and While Loop structures can index and accumulate arrays at their boundaries automatically. These capabilities collectively are called *auto-indexing*. When you enable auto-indexing and wire an array of any dimension from an external node to an input tunnel on the loop border, components of that array enter the loop, one at a time, starting with the first component. The loop indexes scalar elements from 1D arrays, 1D arrays from 2D arrays, and so on. The opposite action occurs at output tunnels—elements accumulate sequentially into 1D arrays, 1D arrays accumulate into 2D arrays, and so on.

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



By default, auto-indexing is disabled for every array wired to a While Loop. Pop up on the array tunnel of a While Loop to enable auto-indexing.



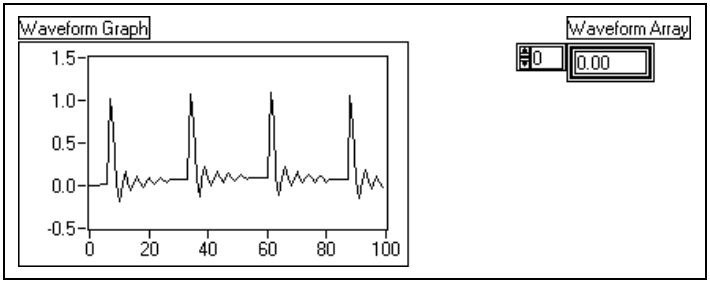
Activity 5-1. Create an Array with Auto-Indexing

Your objective is to create an array using the auto-indexing feature of a For Loop and plot the array in a waveform graph.

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

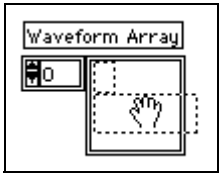
Front Panel

1. Open a new front panel.



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

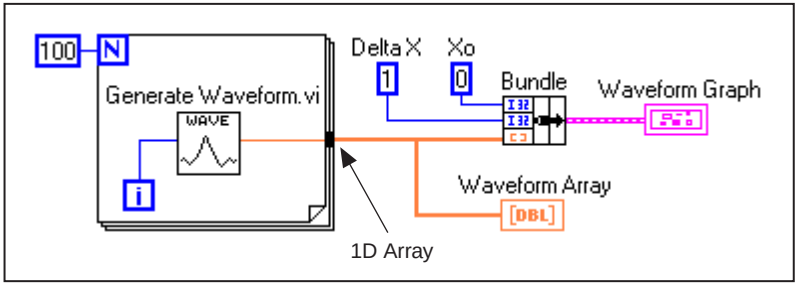
3. Place a digital indicator from **Controls»Numeric** inside the element display of the array shell, as the following illustration shows. This indicator displays the array contents.



4. Place a waveform graph from **Controls»Graph** in the front panel. Label the graph `Waveform Graph`.
5. Enlarge the graph by dragging a corner with the Resizing cursor.
6. Hide the legend and palette.
7. Disable autoscaling by popping up on the graph and deselecting **Y Scale»Autoscale Y**.
8. Use the Text tool to rescale the Y axis to range from `-0.5` to `1.5`.

Block Diagram

9. Build the block diagram shown in the following illustration.



Generate Waveform VI (**Functions»Select a VI…** from the `LabVIEW\Activity` directory)—Returns one point of a waveform. The VI requires a scalar index input, so wire the loop iteration terminal to this input.

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

The For Loop automatically accumulates the arrays at its boundary. This is called auto-indexing. In this case, the numeric constant wired to the loop

count numeric input has the For Loop create a 100-element array (indexed 0 to 99).

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

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

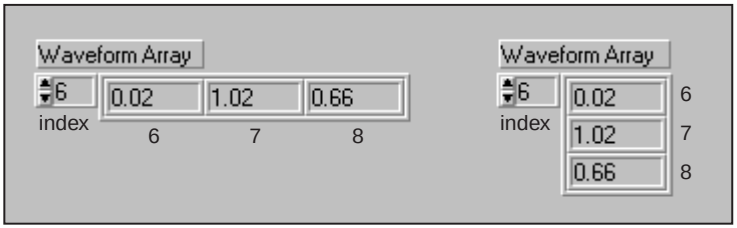
10. From the front panel, run the VI. The VI plots the auto-indexed waveform array on the waveform graph. The initial *X* value is 0 and the delta *X* value is 1.
11. Change the delta *X* value to 0.5 and the initial *X* value to 20. Run the VI again.

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

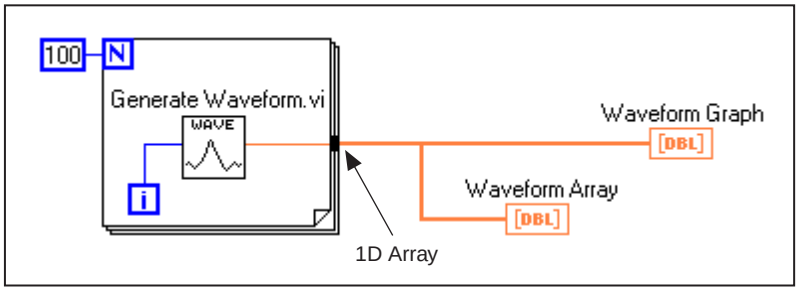
12. You can view any element in the waveform array by entering the index of that element in the index display. If you enter a number greater than the array size, the display dims, indicating that you do not have a defined element for that index.

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

element corresponding to the specified index, as the following illustration shows.



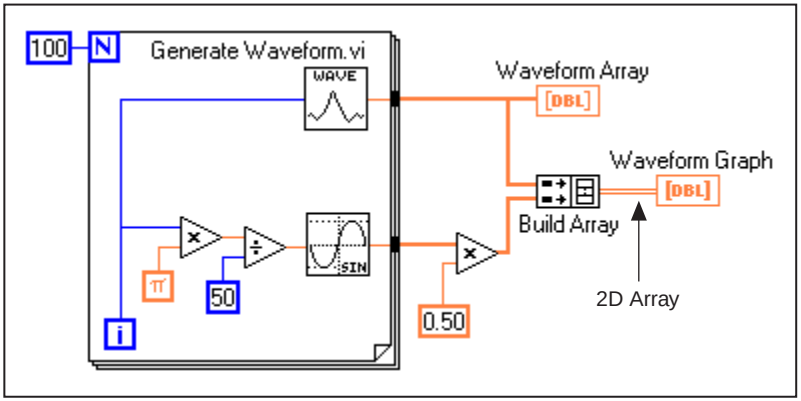
In the previous block diagram, you specified an initial *X* and a delta *X* value for the waveform. The default initial *X* value is zero and the delta *X* value is 1. So, you can wire the waveform array directly to the waveform graph terminal without the initial *X* and delta *X* specified, as the following illustration shows.



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

Multiplot Graphs

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



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

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

Build Array function (**Functions»Array**)—In this exercise, you use this function to create the proper data structure to plot two arrays on a waveform graph, which in this case is a 2D array. Enlarge the Build Array function to create two inputs by dragging a corner with the Positioning tool.

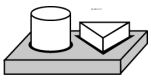
Pi constant (**Functions»Numeric»Additional Numeric Constants**)—Remember that you can find the Multiply and Divide functions in **Functions»Numeric**.

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

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

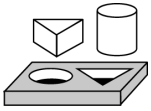
Note ***You can change the appearance of a plot on the graph by popping up in the legend for a particular plot. For example, you can change from a line graph to a bar graph by choosing*** **Common Plots»Bar Graph.**

17. Save the VI as `Graph Waveform Arrays.vi` in the `LabVIEW\Activity directory`.



End of Activity 5-1.

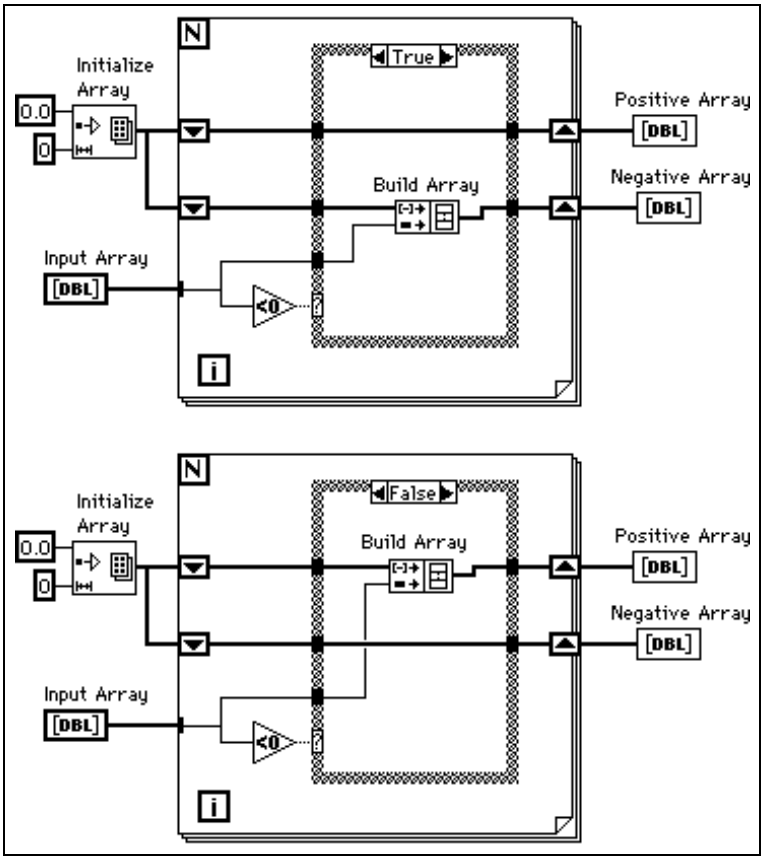
In the previous example, the For Loop executed 100 times because a constant of `100` was wired to the count terminal. The following activity illustrates another means of determining how many times a loop will execute.



Activity 5-2. Use Auto-Indexing on Input Arrays

Your objective is to open and operate a VI that uses auto-indexing in a For Loop to process an array.

1. Open the Separate Array Values VI by selecting **File»Open…**. The VI is located in `Examples\General\arrays.llb`.
2. Open the block diagram. The following illustration shows the block diagram with both TRUE and FALSE cases visible.

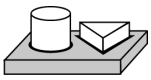


Notice that the wire from `Input Array` changes from a thick wire outside the For Loop, indicating it is an array, to a thin wire inside the loop, indicating it is a single element. The i^{th} element of the array is indexed automatically from the array during each iteration.

Using Auto-Indexing to Set the For Loop Count

Notice that the count terminal is left unwired. When you use auto-indexing on an array entering a For Loop, the loop executes according to the size of the array, eliminating the need to wire a value to the count terminal. If you use auto-indexing for more than one array, or if you set the count in addition to auto-indexing an array, the actual number of iterations is the smallest number possible.

3. Run the VI. Of the eight input values, you will see four in the Positive Array and four in the Negative Array.
4. From the block diagram, wire a constant of 5 to the count terminal of the For Loop. Run the VI. You will see three values in the Positive Array and two in the Negative Array, even though the input array still has eight elements. This demonstrates that if N is set and you are auto-indexing, the smaller number is used for the actual number of iterations of the loop.
5. Close the VI and do not save changes.

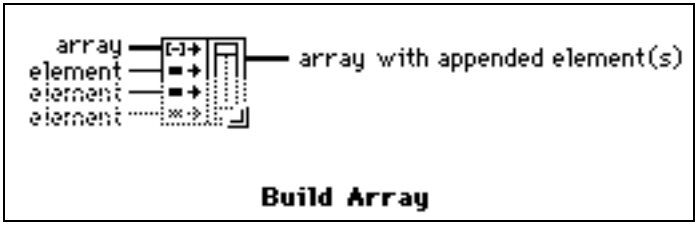


End of Activity 5-2.

Using Array Functions

G has many functions to manipulate arrays located in **Functions»Array**. These functions include Replace Array Element, Search 1D Array, Sort 1D Array, Reverse 1D Array, and Multiply Array Elements. For more information about arrays and the array functions available, refer to Chapter 14, *Array and Cluster Controls and Indicators*, in the *G Programming Reference Manual* or **Online Reference»Function and VI Reference**.

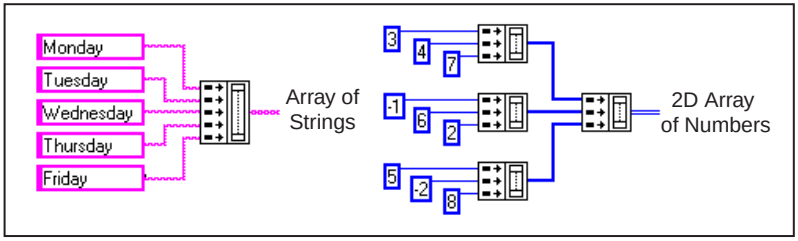
Build Array



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

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

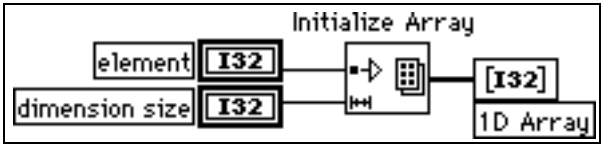
The following illustration shows two ways to create and initialize arrays with values from block diagram constants. On the left, five string constants are built into a 1D array of strings. On the right, three groups of numeric constants are built into three, 1D numeric arrays. Then, the three arrays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



You also can create an array by combining other arrays along with scalar elements. For example, suppose you have two arrays and three scalar elements that you want to combine into a new array with the order array 1, scalar 1, scalar 2, array 2, and scalar 3.

Initialize Array

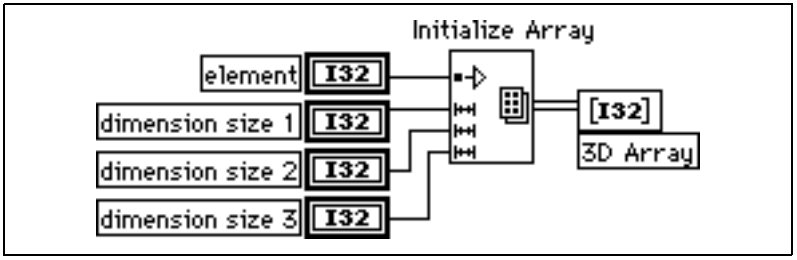
Use this function to create an array whose elements all have the same value. In the following illustration, this function creates a 1D array.



The element input determines the data type and the value of each element. The dimension size input determines the length of the array. For example, if `element` is a long integer with the value of five and `dimension size` has a value of 100, the result is a 1D array of 100 long integers all set to five. You can wire the inputs from front panel control terminals, as shown in the preceding illustration, from block diagram constants, or from calculations on other parts of your diagram.

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

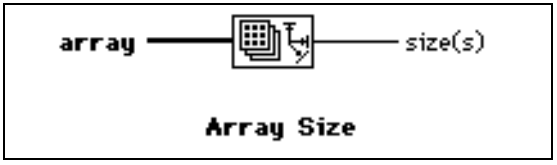
The following block diagram shows how to initialize a 3D array.

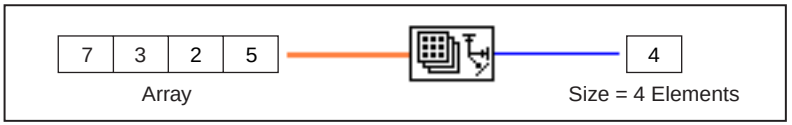


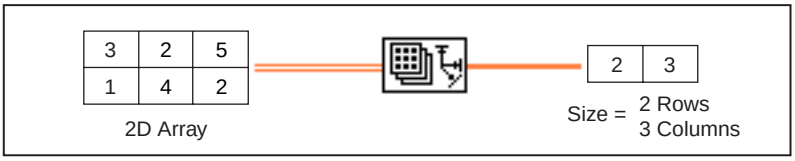
If all the dimension size inputs are zero, the function creates an empty array of the specified type and dimension.

Array Size

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

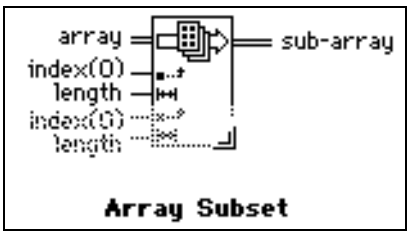




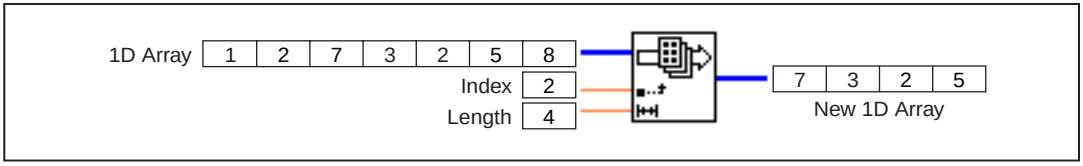


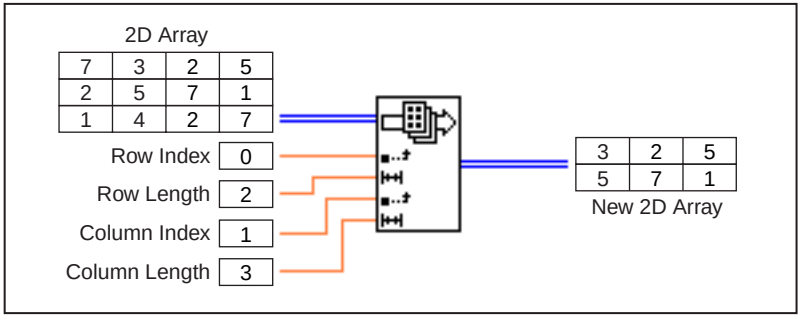
Array Subset

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



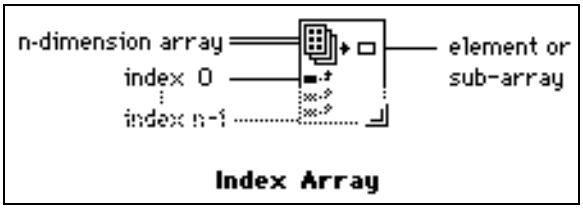
Array Subset returns a portion of an array starting at index and containing length elements. The following illustrations show examples of Array Subsets. Notice that the array index begins with 0.



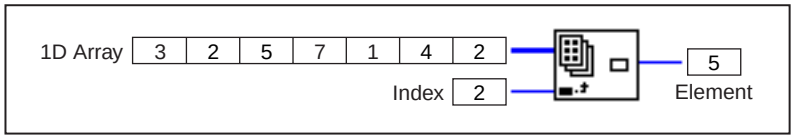


Index Array

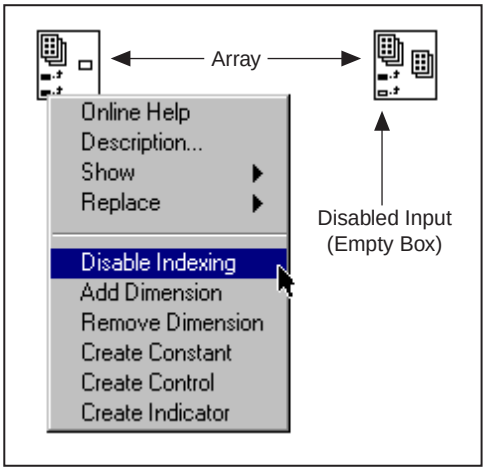
The Index Array function accesses an element of an array.



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

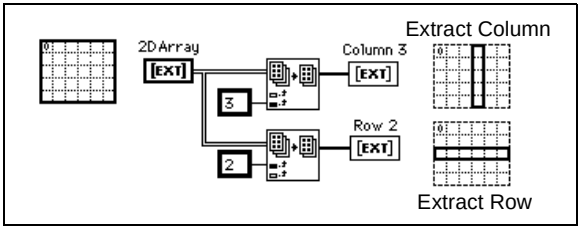


You also can use this function to slice off one or more dimensions of a multi-dimensional array to create a subarray of the original. To do this, stretch the Index Array function to include two index inputs, and select the **Disable Indexing** command on the pop-up menu of the second index terminal as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

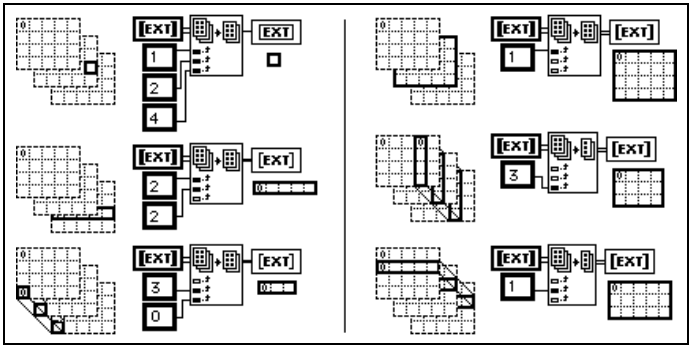


Notice that the index terminal symbol changes from a solid to an empty box when you disable indexing. To restore a disabled index, use the **Enable Indexing** command from the same menu.

You can extract subarrays along any combination of dimensions. The following illustration shows how to extract a 1D row or column arrays from a 2D array.



From a 3D array, you can extract a 2D array by disabling two index terminals, or a 1D array by disabling a single index terminal. The following figure shows several ways to slice a 3D array.



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

- The dimension of the output object must equal the number of disabled index terminals. For example:
  - Zero disabled = scalar element
  - One disabled = 1D component
  - Two disabled = 2D component
- The values wired to enabled terminals must identify the output elements.

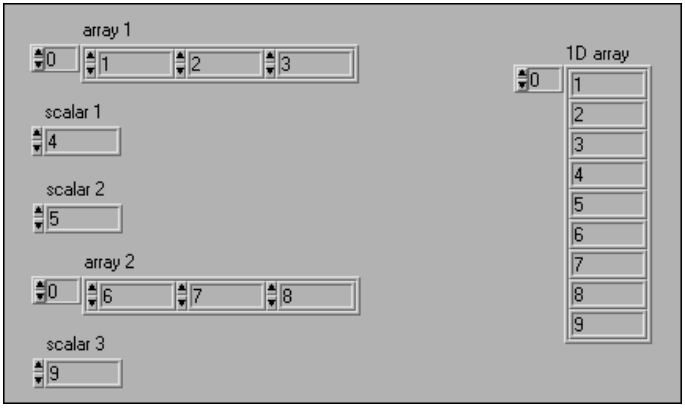
Thus, you can interpret the lower left preceding example as a command to generate a 1D array of all elements at column 0 and row 3. You can interpret the upper right example as a command to generate a 2D array of page 1. The new, 0^{th} element is the one closest to the original, as shown in the preceding illustration.

Activity 5-3. Use the Build Array Function

Your objective is to use the Build Array function to combine elements and arrays into one bigger array.

Front Panel

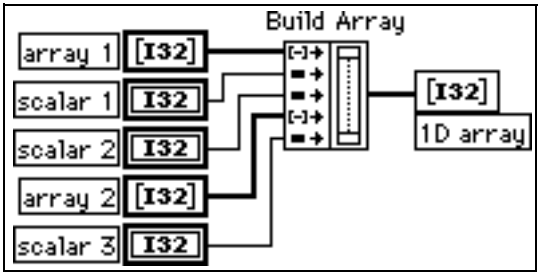
1. Create a new front panel, as shown in the following illustration.



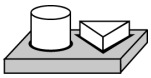
2. Place a digital control from the **Controls»Numeric** palette and label it `scalar 1`. Change its representation to `I32`.
3. Copy and paste it to create two other digital controls and label them `scalar 2` and `scalar 3`.
4. Create an array of digital controls and label it `array 1`. Copy and paste it and label it `array 2`.
5. Expand the arrays and enter the values 1 through 9 in `array 1`, `scalar 1`, `scalar 2`, `array 2`, and `scalar 3`, as shown in the illustration above.
6. Copy the array and paste it and change it to an indicator. Label it `1D array`. Expand it to show nine values.

Block Diagram

7. Place a Build Array function (**Functions»Array**) on the block diagram. Expand it with the Positioning tool to have five inputs.
8. Pop up on the first input in the Build Array node and select **Change to Array**. Do the same for the fourth input.
9. Wire the arrays and scalars to the node. The output array is a 1D array composed of the elements of `array 1` followed by `scalar 1`, `scalar 2`, and the elements of `array 2` and `scalar 3`, as the following illustration shows.



10. Run the VI. You can see the values in `scalar 1`, `scalar 2`, `scalar 3`, `array 1`, and `array 2` appear in a single 1D array.
11. Save the VI as `Build Array.vi` in the `LabVIEW\Activity` directory.



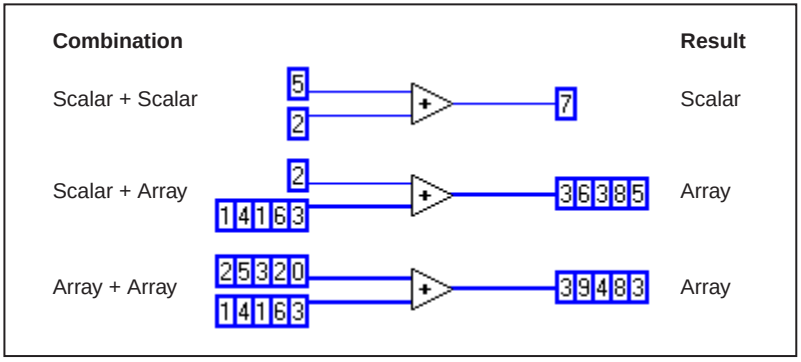
End of Activity 5-3.

Efficient Memory Usage: Minimizing Data Copies

To save memory, you can use single-precision arrays instead of double-precision arrays. For information about how memory is allocated, see the section *Monitoring Memory Usage* in Chapter 28, *Performance Issues*, in the *G Programming Reference Manual*.

What is Polymorphism?

Polymorphism is the ability of a function to adjust to input data of different types, dimensions, or representations. Most G functions are polymorphic. For example, the following illustrations show some of the polymorphic combinations of the Add function.



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

You can apply these principles to other G functions and data types. G functions are polymorphic to different degrees. Some functions might accept numeric and Boolean inputs, others might accept a combination of any other data types. For more information about polymorphism, see **Online Reference»Function and VI Reference**.

Clusters

A cluster is a data type that can contain data elements of different types. The cluster in the block diagram that you will build in Activity 5-4 groups related data elements from multiple places on the block diagram, reducing wire clutter. When you use clusters, your subVIs require fewer connection terminals. A cluster is analogous to a record in Pascal or a struct in C. You can think of a cluster as a bundle of wires, much like a telephone cable. Each wire in the cable would represent a different element of the cluster. The components include the initial *X* value (0), the delta *X* value (1), and the Y array (waveform data, provided by the numeric constants on the block diagram). In G, use the Bundle function to assemble a cluster. For more information about Clusters refer to Chapter 14, *Array and Cluster Controls and Indicators*, in the *G Programming Reference Manual*.

Graphs

A *graph* is a two-dimensional display of one or more data arrays called plots. There are three types of graphs in the **Controls»Graph** palette:

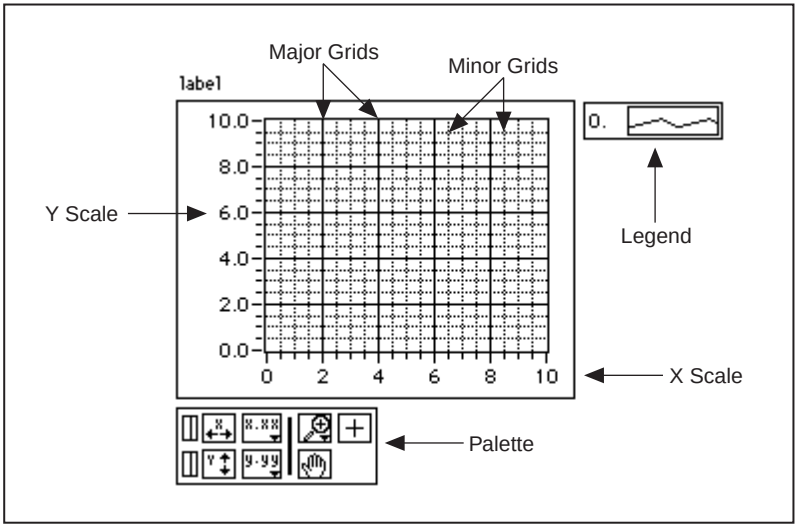
- XY graph
- Waveform graph
- Intensity graph

The difference between a graph and a chart is that a graph plots data as a block, whereas a chart plots data point by point, or array by array.

For examples of graph VIs, see `Examples\General\Graphs`.

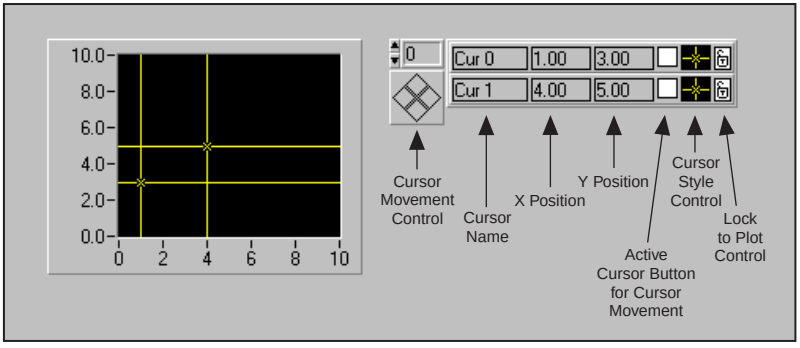
Customizing Graphs

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



Graph Cursors

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



For more detailed information on customizing graphs, see Chapter 15, *Graph and Chart Controls and Indicators*, in the *G Programming Reference Manual*.

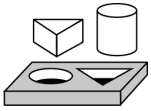
Refer to the ZoomGraph VI in `Examples\General\Graphs\zoom.llb` for an example that reads cursor values and programmatically zooms in and out of a graph using the cursors.

Graph Axes

You can format the scales of a graph to represent either absolute or relative time. Use absolute time format to display the time, date, or both for your scale. If you do not want G to assume a date, use relative time format. To select absolute or relative time format, pop up on the chart and select the scale you want to modify. Select **Formatting…**. This enables the **Formatting** dialog box, which you can use to specify different attributes of the chart.

Data Acquisition Arrays

Data returned from a plug-in data acquisition board using the Data Acquisition VIs can be in the form of a single value, a 1D array, or a 2D array. You can find a number of graph examples located in `Examples\General\Graphs`, which contains VIs to perform varied functions with arrays and graphs.

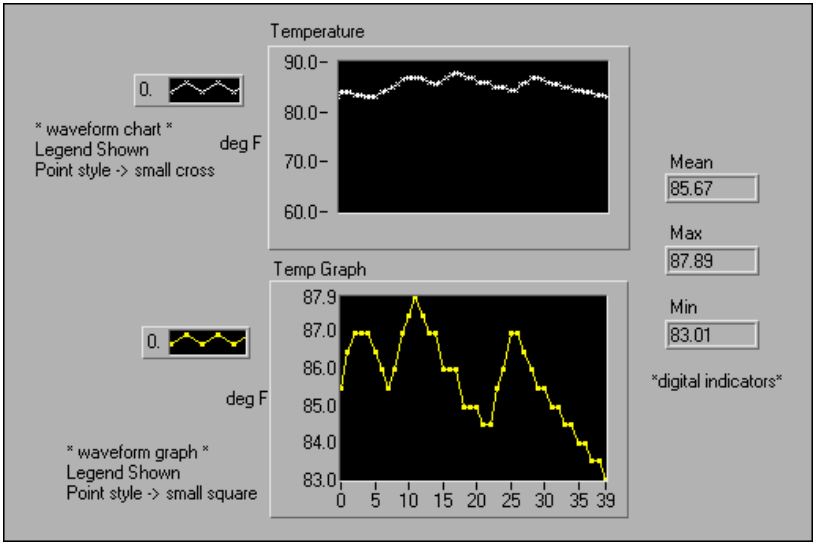


Activity 5-4. Use the Graph and Analysis VIs

Your objective is to build a VI that measures temperature and displays the values in real time. It also displays the average, maximum, and minimum temperatures.

Front Panel

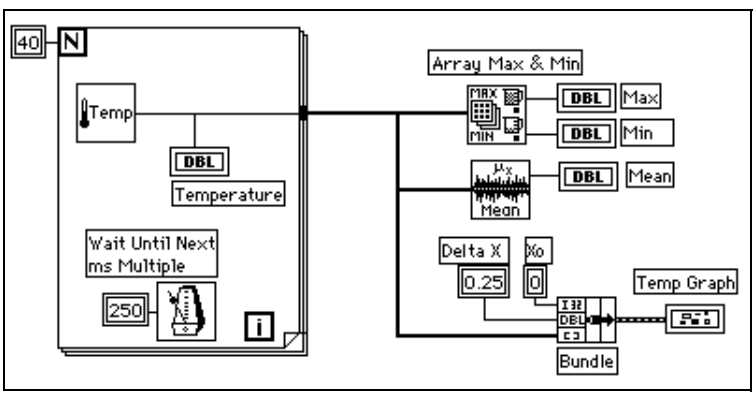
1. Create a new front panel as shown in the following illustration. You can modify the point styles of the waveform chart and waveform graph by popping up on their legends. Scale the charts as shown.



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

Block Diagram

2. Build the block diagram as shown in the following illustration:



Digital Thermometer VI (**Functions»Select a VI** from the `LabVIEW\Activity` directory)—Returns one temperature measurement.

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

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

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

Mean VI (**Functions»Analysis»Probability and Statistics** or **Functions»Base Analysis** for LabVIEW Base Package users)—Returns the average of the temperature measurements.

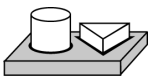
Bundle function (**Functions»Cluster**)—Assembles the plot components into a cluster. The components include the initial *X* value (0), the delta *X* value (0.25), and the Y array (temperature data). Use the Positioning tool to resize the function by dragging one of the corners.

The For Loop executes 40 times. The Wait Until Next ms Multiple function causes each iteration to take place every 250 milliseconds. The VI stores the temperature measurements in an array created at the For Loop border (auto-indexing). After the For Loop completes execution, the array is passed on to the subVIs and `Temp Graph`.

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

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

3. Return to the front panel and run the VI.
4. Save the VI as `Temperature Analysis.vi` in the `LabVIEW\Activity` directory.



End of Activity 5-4.

Intensity Plots

LabVIEW has two methods for displaying three-dimensional data: the intensity chart and the intensity graph. Both intensity plots accept two-dimensional arrays of numbers, where each number is mapped to a color. You can define the color mapping interactively, using an optional color ramp scale, or programmatically, using an attribute node for the chart. For examples using the intensity chart and graph, refer to `intgraph.llb` in the `Examples\General\Graphs` directory.

6

Strings and File I/O

This chapter introduces string controls and indicators and file input and output operations and provides activities that illustrate how to accomplish the following:

- Create string controls and indicators
- Use string functions
- Perform file input and output operations
- Save data to files in spreadsheet format
- Write data to and read data from text files

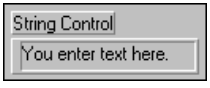
Strings

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

For examples of strings, see `Examples\General\strings.llb`.

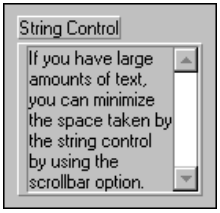
Creating String Controls and Indicators

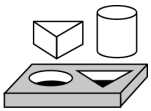
You can find the string control and indicator, shown at left, in **Controls»String & Table**. You can enter or change text inside a string control using the Operating tool or the Labeling tool. Enlarge string controls and indicators by dragging a corner with the Positioning tool.



Strings and File I/O

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



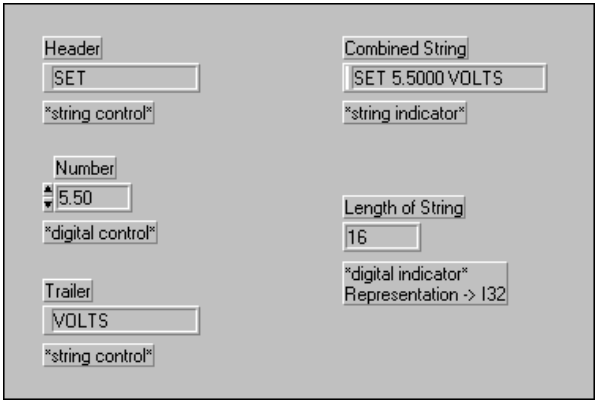


Activity 6-1. Concatenate a String

LabVIEW has many functions to manipulate strings. For this activity, your objective is to use some of the string functions to convert a number to a string and concatenate that string with other strings to form a single output string.

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

Front Panel

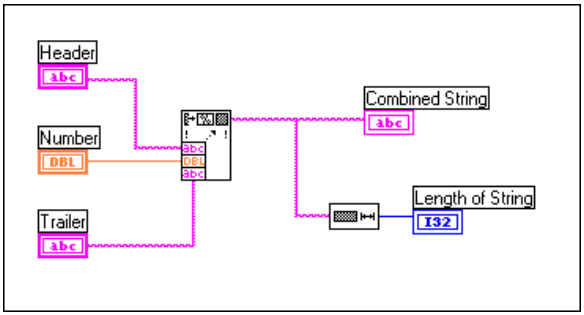


The two string controls and the digital control can be combined into a single output string and displayed in the string indicator. The digital indicator displays the string length.

The Combined String output in this activity has a similar format to command strings used to communicate with GPIB (IEEE 488) and serial (RS-232 or RS-422) instruments. Refer to Part II, *I/O Interfaces*, of this manual to learn more about strings used for instrument commands.

Block Diagram

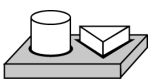
2. Build the block diagram shown in the following illustration.



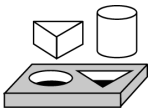
Format Into String function (**Functions»String**) concatenates and formats numbers and strings into a single output string. Use the Resizing cursor on the icon to add three argument inputs.

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

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



End of Activity 6-1.



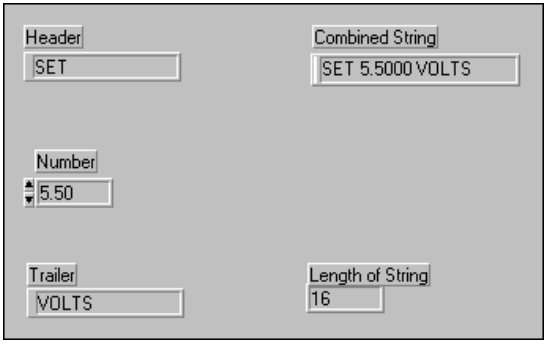
Activity 6-2. Use Format Strings

Your objective is to create a string according to the format you specify.

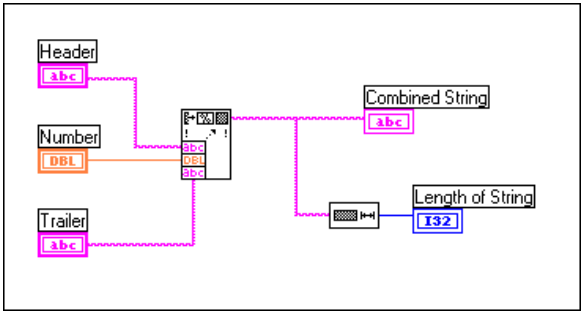
You will use the Build String VI that you created in Activity 6-1 to create a format string. With format strings, you can specify the format of arguments, including the field width, base (hex, octal, and so on), and any text that separates the arguments.

Front Panel

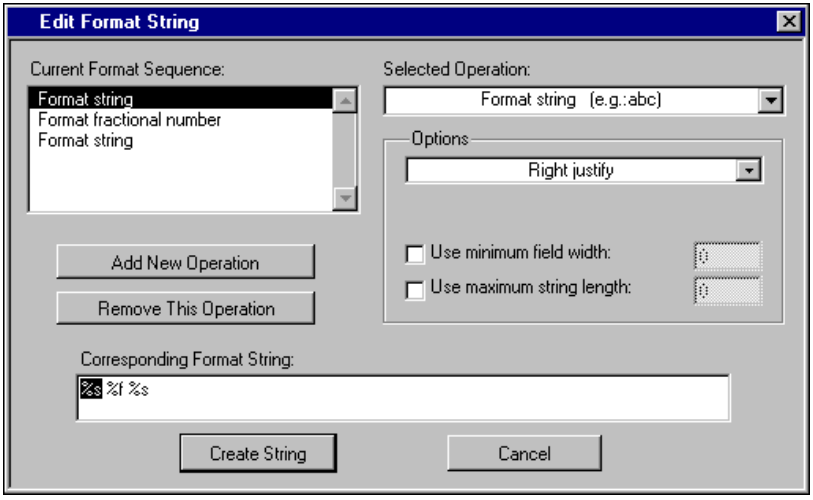
1. Open the Build String VI that you created in Activity 6-1.



Block Diagram



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

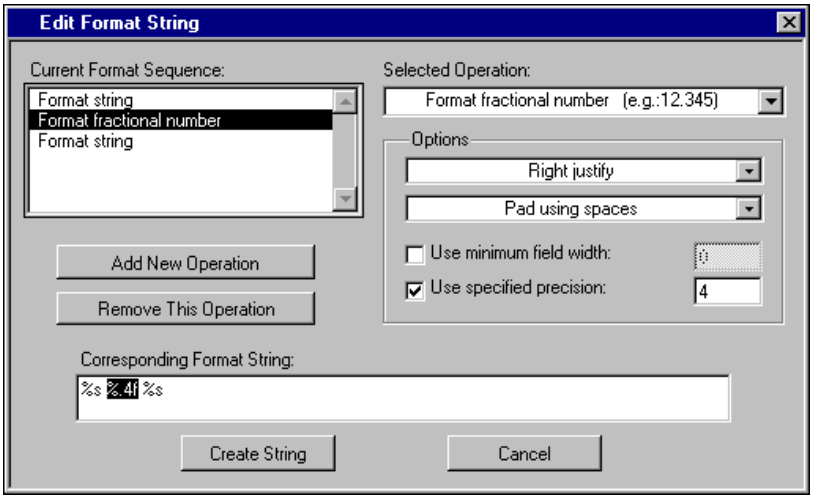


☞ **Note** ***You can also double-click on the node to access the Edit Format String dialog box.***

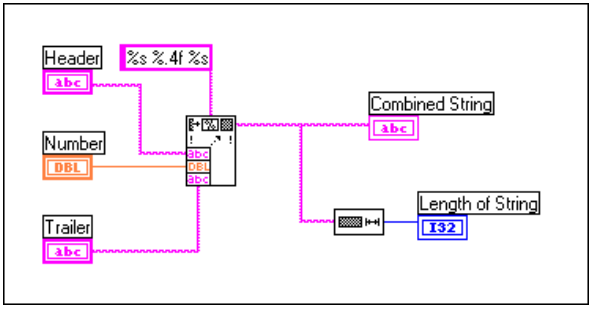
Notice that the Current Format Sequence contains the argument types, in the order that you wired them.

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

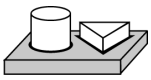
illustration shows the selected options to set the precision of number.



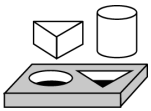
4. Press the **Create String** button. Pressing this button automatically inserts the correct format string information and wires format string to the function, as shown in the following illustration.



5. Return to the front panel and type text inside the two string controls and a number inside the digital control. Run the VI.
6. Save and close the VI. Name it `Format String.vi`.



End of Activity 6-2.

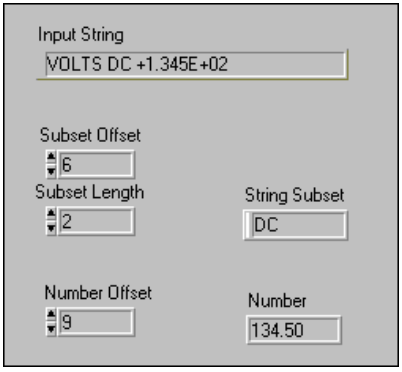


Activity 6-3. String Subsets and Number Extraction

Your objective is to take a subset of a string that contains the string representation of a number and convert it to a numeric value.

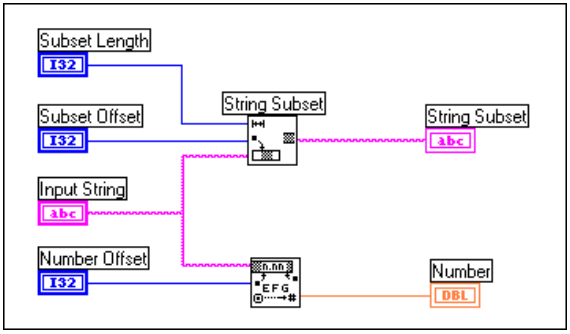
Front Panel

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



Block Diagram

2. Open the block diagram of the Parse String VI, shown in the following illustration.



LabVIEW uses the String Subset and Scan From String functions to parse the input string.

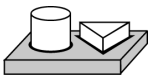
String Subset function (**Functions»String**) returns the substring beginning at **offset** and contains **length** number of characters. The first character offset is zero.

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

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

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

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



End of Activity 6-3.

File I/O

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

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

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

This section discusses ASCII byte stream files because that is the most common data file format. For examples of file I/O, see `Examples\File`.

File I/O Functions

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

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

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

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

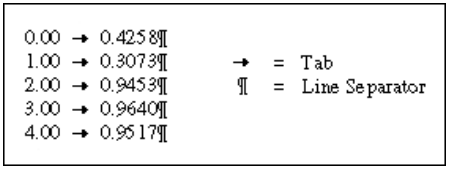
The Write To Spreadsheet File VI converts a 1D or 2D array of single-precision numbers to a text string and writes the string to a new byte stream file or appends the string to an existing file. You can optionally transpose the data. This VI opens or creates the file beforehand and closes it afterwards.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

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

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

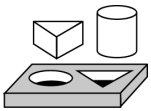
Writing to a Spreadsheet File

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



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

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

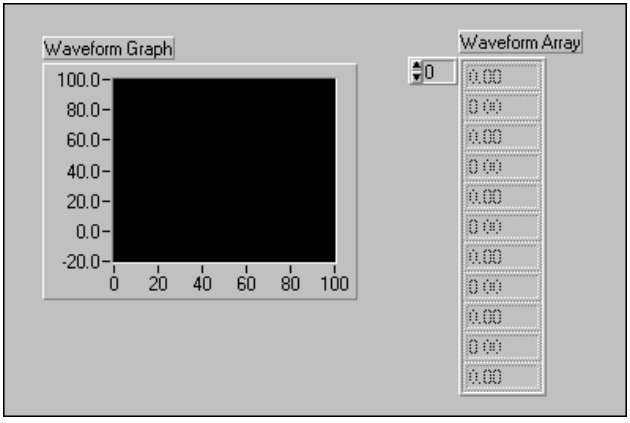


Activity 6-4. Write to a Spreadsheet File

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

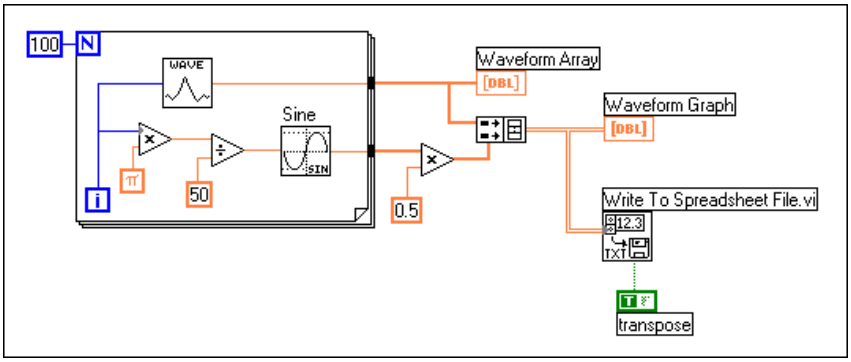
Front Panel

1. Open the `Graph Waveform Arrays.vi` you built in Activity 5-1. As you recall, this VI generates two data arrays and plots them on a graph. You modify this VI to write the two arrays to a file where each column contains a data array.



Block Diagram

2. Open the block diagram of `Graph Waveform Arrays.vi` and modify the VI by adding the block diagram functions that have been added to the lower right of the following illustration.



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

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

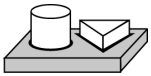
3. Return to the front panel and run the VI. After the data arrays have been generated, a file dialog box prompts you for the file name of the new file you are creating. Type in a file name and click on **OK**.

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

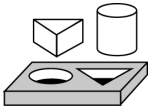
4. Save the VI, name it `Waveform Arrays to File.vi`, and close the VI.
5. You now can use spreadsheet software or a text editor to open and view the file you just created. You should see two columns of 100 elements.

In this example, the data was not converted or written to file until the entire data arrays had been collected. If you are acquiring large buffers of data or

would like to write the data values to disk as they are being generated, then you must use a different File I/O VI.



End of Activity 6-4.

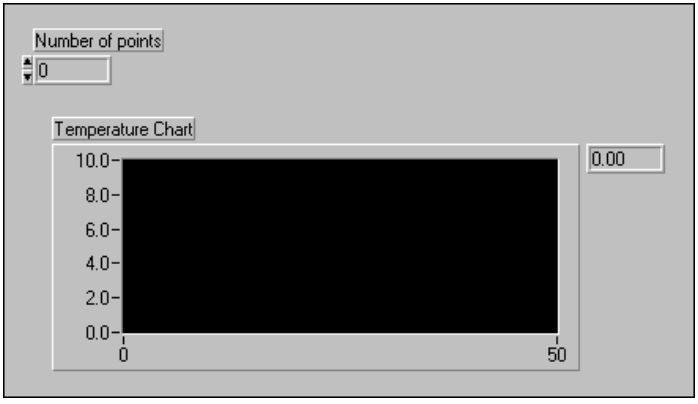


Activity 6-5. Append Data to a File

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

Front Panel

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

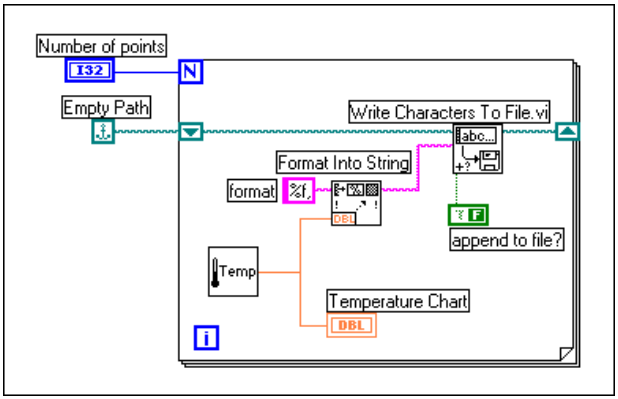


The front panel contains a digital control and a waveform chart. Select **Show»Digital Display**. The `# of points` control specifies how many temperature values to acquire and write to file. The chart displays the temperature curve. Rescale the Y axis of the chart for the range 70.0 to 90.0, and rescale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

2. Pop up on the # of points digital control and choose **Representation»I32**.

Block Diagram

3. Open the block diagram.



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

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

Digital Thermometer VI returns a simulated temperature measurement from a temperature sensor.

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

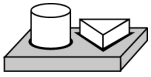
String constant (**Functions»String**). This format string specifies that you want to convert a number to a fractional format string and follow the string with a comma.

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

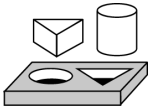
Boolean Constant (**Functions»Boolean**) sets the `append to file?` input of the Write Characters To File VI to True so that the new temperature

values are appended to the selected file as the loop iterates. Click the Operating tool on the constant to set its value to True.

7. Return to the front panel and run the VI with the `# of points` set to 20. A file dialog box prompts you for a file name. When you enter a file name, the VI starts writing the temperature values to that file as each point is generated.
8. Save the VI as `Write Temperature to File.vi` in the `LabVIEW\Activity` directory.
9. Use any word processing software such as Write for Windows, Teach Text for Macintosh, or a text editor in UNIX to open that data file and view the contents. You should get a file containing twenty data values (with a precision of three places after the decimal point) separated by commas.



End of Activity 6-5.

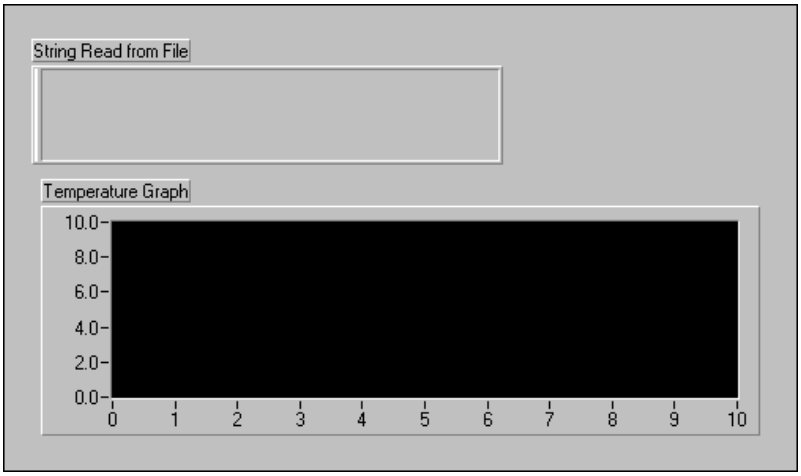


Activity 6-6. Read Data from a File

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

Front Panel

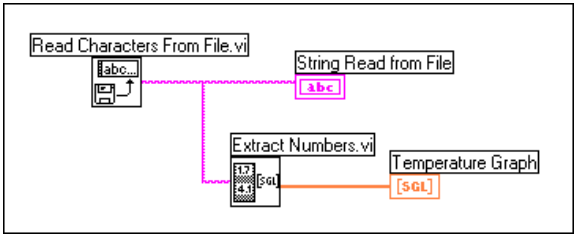
1. Open a new front panel and build the front panel shown in the following illustration.



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

Block Diagram

2. Build the block diagram as shown in the following illustration.

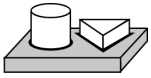


The Read Characters From File VI (**Functions»File I/O**) reads the data from the file and outputs the information in a string. If no path name is specified, a file dialog box prompts you to enter a file name. In this example, you do not need to determine the number of characters to read because there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

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

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

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



End of Activity 6-6.

Using the File I/O Functions

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

Specifying a File

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

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

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

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

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

Paths and Refnums

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

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

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

File I/O Examples

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

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

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

Datalog Files

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

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

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

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

Part II

I/O Interfaces

This section contains basic information on the interfaces to which you can input and output data, which are data acquisition, GPIB, serial, and VXI. Refer to the *Data Acquisition Basics Manual* for introductory information on real-time data acquisition. VISA (Virtual Instrument Software Architecture) is a single software API that interfaces with GPIB, serial, and VXI instruments. LabVIEW applications developed especially for a specific instrument are called instrument drivers. National Instruments provides several instrument drivers using the VISA library, but you can also build your own instrument drivers.

Part II, *I/O Interfaces*, contains the following chapters.

- Chapter 7, *Getting Started with a LabVIEW Instrument Driver*, explains how to create and use National Instruments instrument drivers.
- Chapter 8, *LabVIEW VISA Tutorial*, shows you how to implement common VISA applications using message-based and register-based communication as well as events and locking.
- Chapter 9, *Introduction to LabVIEW GPIB Functions*, explains how the GPIB operates and the difference between the IEEE 488 and IEEE 488.2 interfaces.
- Chapter 10, *Serial Port VIs*, describes the VIs for serial port communication and explains the important factors that affect serial communication.

7

Getting Started with a LabVIEW Instrument Driver

This chapter begins by describing how to install and use instrument drivers from the Instrument Driver Library, and ends with instruction on creating your own instrument driver. This chapter steps you through common techniques for verifying communication with your instrument, developing an application using instrument drivers, and creating an instrument driver.

What is a LabVIEW Instrument Driver?

An instrument driver is a set of LabVIEW VIs that communicate with an instrument using LabVIEW's standard VISA I/O functions. Each VI corresponds to a programmatic operation, such as configuring, reading from, writing to, and triggering an instrument. LabVIEW instrument drivers eliminate the need to learn the complex, low-level programming commands for each instrument.

The LabVIEW instrument driver library contains instrument drivers for a variety of programmable instruments that use the GPIB, VXI or serial interface. You can use a library driver for your instrument as is. However, instrument drivers are distributed with their block diagram source code, so you can customize them for your specific application if need be.

Where Can I Get Instrument Drivers?

Instrument drivers can be installed from an instrument driver CD or downloaded from the National Instruments web site. You can obtain the latest instrument driver order form by using a touch-tone phone to call the National Instruments automated fax system, Fax-on-Demand, at (512) 418-1111 or (800) 329-7177. Or you can download drivers using the Instrument Driver Network on the web. To access this Network, connect to `http://www.natinst.com/idnet`.

If an instrument driver for your particular instrument does not exist, you can:

1. Try using a driver for a similar instrument. Often similar instruments from the same manufacturer have similar if not identical command sets.
2. Create an instrument driver using the guidelines in the *Developing a Quick and Simple LabVIEW Instrument Driver* section in this chapter.
3. Develop a complete, fully functional instrument driver. To develop a National Instruments quality driver, you can download Application Note 006, *Developing a LabVIEW Instrument Driver*, from our web site. This application note will help you to develop a complete instrument driver.

Where Should I Install My LabVIEW Instrument Driver?

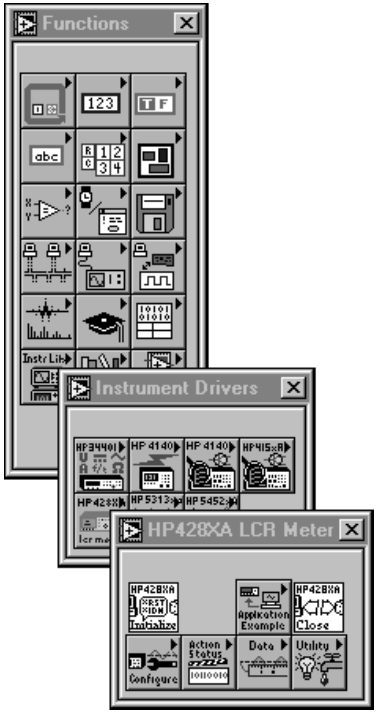
Instrument drivers should be installed as a subdirectory of your `LabVIEW/instr.lib`. For example, the HP34401A instrument driver, which is included with LabVIEW, is installed in the following directory:

```
Labview/instr.lib/hp34401a
```

Within this directory you will find the menu files and VI libraries that make up an instrument driver. The menu files allow you to view your instrument driver VIs from the **Functions** palette. The VI libraries contain the instrument driver VIs.

How Do I Access the Instrument Driver VIs?

You can find the Instrument Driver VIs near the bottom of the **Functions** palette in the **Instrument Drivers** subpalette.



Many of the instrument drivers have menus palettes which have the following components:

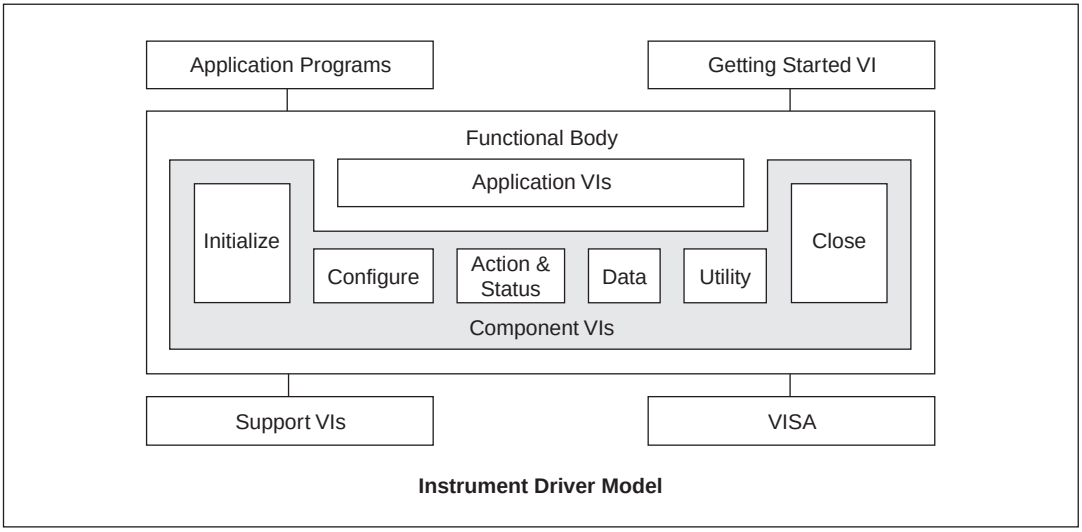
- Initialize VI
- Close VI
- Application Example Subpalette
- Configuration Subpalette
- Action/Status Subpalette
- Data Subpalette
- Utility Subpalette

Instrument driver VIs can also be accessed using the **Select a VI** option from the **Functions** palette. To view the entire instrument driver hierarchy

you might open the VI Tree VI. This is a non-executable VI that is designed to show the functional structure of the instrument driver.

Instrument Driver Structure

The following figure shows the organization of a standard instrument driver. Once you understand this model, you will find it applies to numerous instrument drivers.

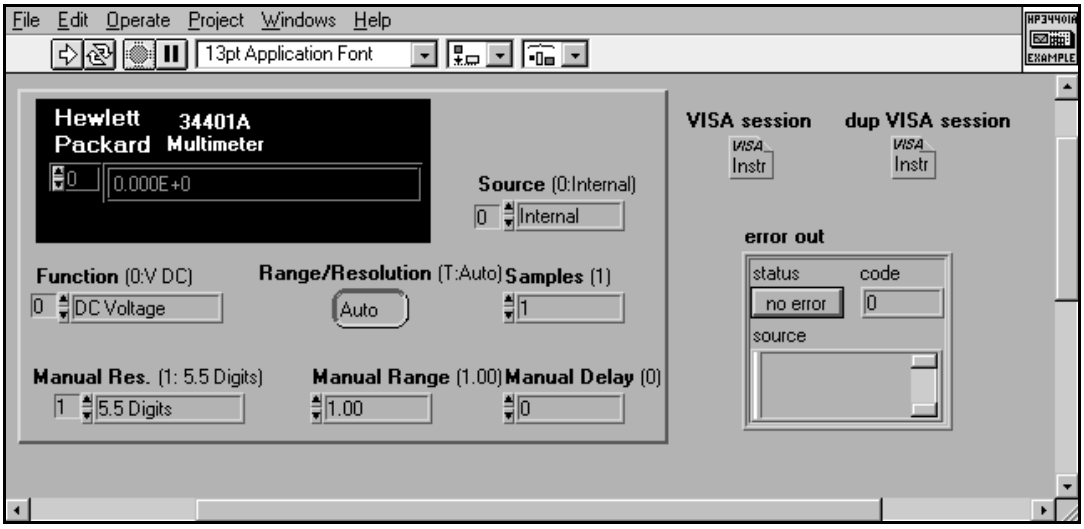


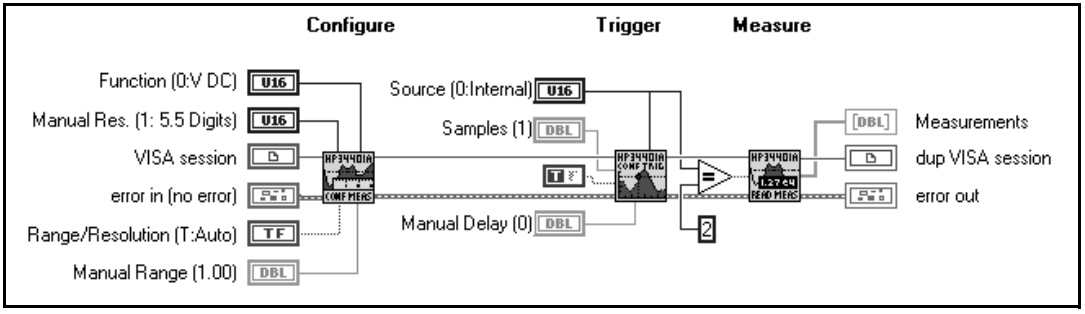
Instrument Driver Model

The Getting Started VIs are simple application VIs you can use without modification. Run this VI to verify communication with your instrument. Typically you will only need to change the instrument address before running the VI from the front panel. However, there are a few that also require you to specify the VISA Resource name (for example, GPIB::2). For more information on VISA Resource names, see Chapter 8, *LabVIEW VISA Tutorial*. The Getting Started VI generally consists of three sub-VIs: the Initialize VI, an Application VI, and the Close VI.

The Application VIs are high-level examples of grouping together low-level component functions to execute a typical programmatic instrument operation. For example, the Application VIs might include VIs to control the most commonly used instrument configurations and measurements. These VIs serve as a code example to execute a common operation such as configuring the instrument, triggering, and taking a measurement.

Because the application VIs are standard VIs with icons and connector panes, you can call them from any high-level application when you want a single, measurement-oriented interface to the driver. For many users, the application VIs are the only instrument driver VIs needed for instrument control. The HP34401 Example VI, shown in the following figure, demonstrates an application VI front panel and block diagram.





The *initialize VI*, the first instrument driver VI called, establishes communication with the instrument. Additionally, it can perform any necessary actions to place the instrument either in its default power on state or in some other specific state. Generally, the *initialize VI* only needs to be called once at the beginning of your application program.

The *configuration VIs* are a collection of software routines that configure the instrument to perform the desired operation. There may be numerous *configuration VIs*, depending on the particular instrument. After these VIs

are called, the instrument is ready to take measurements or stimulate a system.

The *action/status* category contains two types of VIs. *Action* VIs initiate or terminate test and measurement operations. These operations can include arming the trigger system or generating a stimulus. These VIs are different from the *configuration VIs* because they do not change the instrument settings, but only order the instrument to carry out an action based on its current configuration. The *status VIs* obtain the current status of the instrument or the status of pending operations.

The *data VIs* transfer data to or from the instrument. Examples include VIs for reading a measured value or waveform from a measurement instrument, VIs for downloading waveforms or digital patterns to a source instrument.

The *utility VIs* perform a variety of operations that are auxiliary to the most often used instrument driver VIs. These VIs include the majority of the instrument driver template VIs such as reset, self-test, revision, error query, and error message and may include other custom instrument driver VIs that perform operations such as calibration or storage and recall of setups.

The *close VI* terminates the software connection to the instrument and frees up system resources. Generally, the *close VI* only needs to be called once at the end of your application program or when you finish communication with your instrument. You should make sure that for each successful call to the initialize VI, that you have a matching close VI—otherwise you will be maintaining unnecessary memory resources.

☞ **Note** ***Application functions do not call initialize and close. To run an application function, you must first run the initialize VI. The Getting Started VI calls initialize and close.***

Obtaining Help for Your Instrument Driver VIs

LabVIEW instrument drivers are documented through the LabVIEW **Help** window. There is a **Help** description for each instrument driver VI as well as for each front panel control. To display the **Help** window, choose **Show Help** from the **Help** menu. To display help for the VI, place the cursor over the VI icon. To display help for the front panel controls, place the cursor over the desired control. If you cannot see the entire description in the **Help** window, you can obtain control or indicator help by selecting **Data Operations»Description...** from the control or indicator pop-up menu.

Running the Getting Started VI Interactively
(Selecting the GPIB Address, Serial Port, and Logical Address)

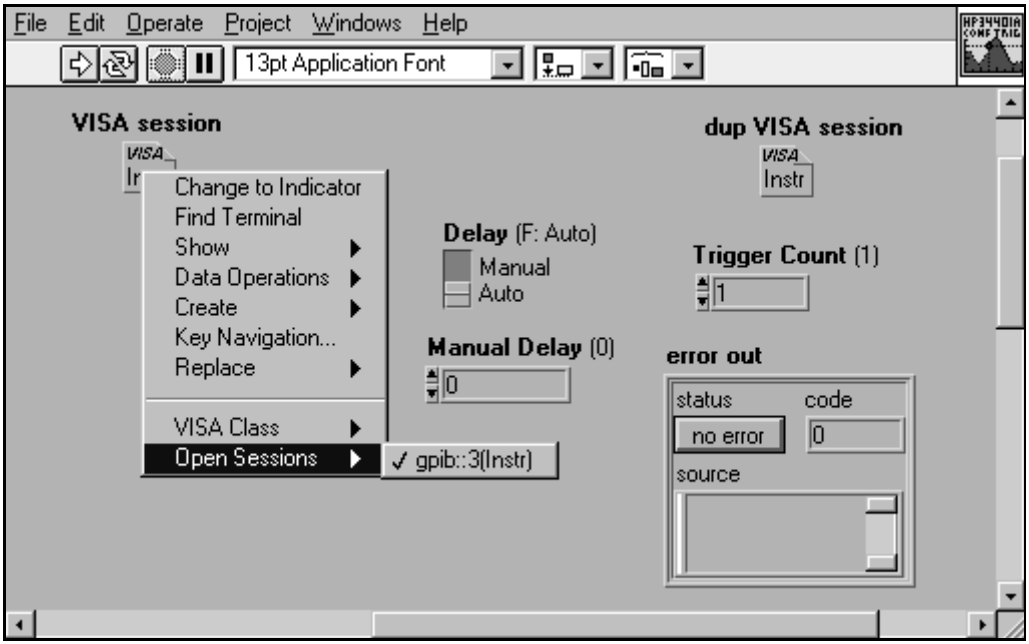
To verify communication with your instrument and test a typical programmatic instrument operation, you should first open the Getting Started VI. Look over each of the controls and set them appropriately. Generally, with the exception of the address field, the defaults for most controls will be sufficient for your first run. You will need to set the address appropriately. If you do not know the address of your instrument, refer to the Instrument Wizard for help. After running the VI, check to see that reasonable data was returned and an error was not reported in the error cluster. The most common reasons for the Getting Started VI to fail are:

1. NI-VISA is not installed. If you did not choose this as an option during your LabVIEW installation, you will need to install it before rerunning your Getting Started VI.
2. The instrument address was incorrect. The Getting Started VI requires you to specify the correct address for your instrument. If you are not certain of your instrument's address, run the Instrument Wizard or the Find Resource function. If you are not familiar with the syntax for the address string, refer to Chapter 8, *LabVIEW VISA Tutorial*, for help. To access the Instrument Wizard, select **Solution Wizard** in the LabVIEW dialog box. When prompted, select **Instrument Wizard**.
3. The instrument driver does not support the exact model you are using. You might need to double-check that the instrument driver supports the instrument model you are using.

Once you have verified basic communication with your instrument using the Getting Started VI, you probably want to customize instrument control for your needs. If your application needs are similar to the Getting Started VI, the simplest means of creating a customized VI is to save a copy of the Getting Started VI by selecting **Save As...** from the **File** menu. You can change the default values on the front panel by selecting **Make Current Values Default** from the **Operate** menu. Block diagram changes might include changing the constants wired to the Application VI or other sub-VIs. As mentioned earlier, the block diagram of the Getting Started VI generally consists of three VIs: The Initialize VI, an Application Function VI, and the Close VI.

Interactively Testing Component VIs

Many users like to test out component VIs interactively before they include them in their application. This helps to select appropriate instrument configuration settings. To run the component VIs from their front panels, you will first need to run the Initialize VI. For subsequent VIs, you will need to first pop-up on the VISA Session control and select your resource name from the **Open Sessions...** sub-menu, as shown below:



Once you have selected your resource. You can interactively run the component VI from the front panel multiple times without resetting the resource selection.

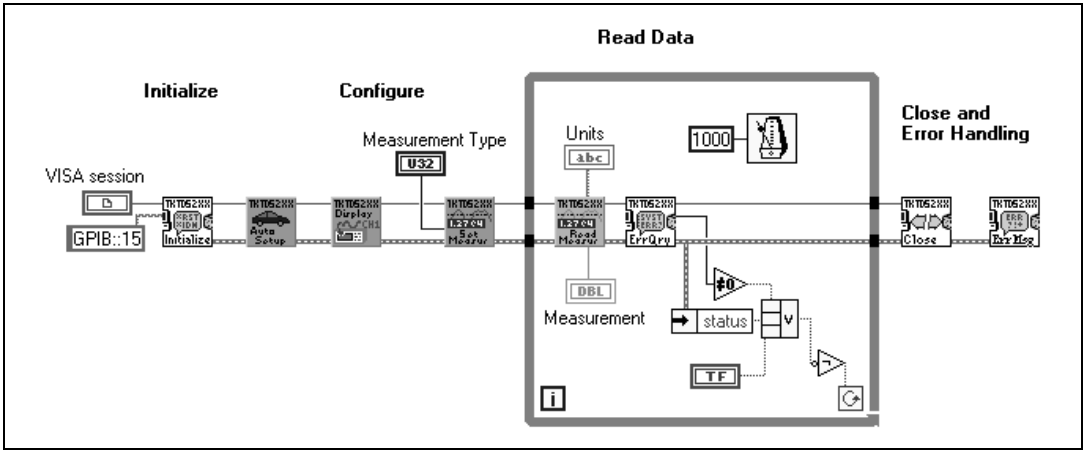
In general, you should run the component VIs in the order you want to call them in your application. First you run the Initialize VI, followed by one or more configuration VIs. If you are using a trigger for your measurement, you might need to call an action VI to arm the trigger. Calls to data VIs then collect the measured value(s). When you are finished testing the instrument driver component VIs, you should run the Close VI to deallocate resources.

Building Your Application

After selecting the component VIs you need and their execution order, you can then build your application. If the execution order is similar to the Application VI, you can modify the Application VI's block diagram. If your application differs significantly from the Application VI, you should build your own VI.

You should place the VIs on your block diagram in order and then wire them together using the VISA session and the error cluster parameters. You do not need to wire all inputs for all component VIs. If the default values are sufficient for your application, you do not need to wire the input terminals. For key inputs, you might want to wire the defaults anyway as a means of documenting your VI. For repeated measurements, you should place your data measurement VIs in a loop. Remember, if you place a component VI inside a loop, you must disable indexing on the VISA Session and Error cluster wires that are passed into and out of the loop.

To check for instrument errors, you should periodically call the Error Query VI. As shown in the following figure, a user could use an oscilloscope to take frequency measurements once per second and display them to the operator. Notice that the loop terminates on three possible conditions: the operator stops the VI using a front panel control, an instrument error is detected by the Error Query VI, or an error occurs with the VISA I/O interface. If an error occurs within the loop, the Error Message VI will then display a popup message to the operator. The Error Message VI is similar to LabVIEW's General Error Handler VI, except that additional instrument-specific errors can be reported. One should use the Error Message VI after executing several instrument driver VIs to recognize and display any errors that may have occurred.



Related Topics

Open VISA Session Monitor VI

The Open VISA Session Monitor VI is handy if you are involved with interactive or programmatic debugging of your instrument driver application. During your debugging you might discover that you have many open VISA sessions that need to be closed. If you open a significant number of the VISA sessions without closing them, you decrease the available memory resources. To close all the open sessions quickly, you can run the Open VISA Session Monitor VI in the `labview/vi.lib/utility/visa.llb` library. Alternatively, you could save your work, exit, and re-enter LabVIEW. Exiting LabVIEW closes all your open VISA sessions.

Error Handling

It is important to perform error handling in instrument control applications because there are several potential sources for errors.

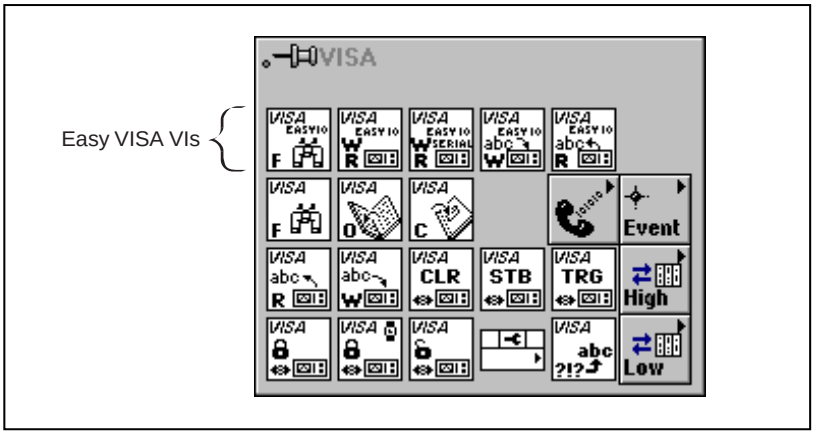
- VISA functions can return errors because VISA or the underlying software or hardware is not properly installed. For example, when communicating with GPIB instruments, NI-488.2 must be installed to correctly to use National Instruments' GPIB controller card. Similarly, if the board is not installed or is not correctly configured, the instrument driver VIs will return an error. This type of error can be detected with the Error Message VI or LabVIEW's General Error Handler VI.
- VISA functions can return errors if the device you are accessing is not responding the commands you have sent. The instrument could be incorrectly addressed, malfunctioning, or unable to understand the commands that are being sent. This type of error can be detected with the Error Message VI or LabVIEW's General Error Handler VI.
- The instrument reports errors. Generally, an instrument will flag an error for reasons ranging from invalid commands to settings out of range to missing hardware options. These instrument errors can be detected by calling the instrument driver's Error Query VI followed by the Error Message VI.

For more information on error handling, see the *Error Handling with VISA* section in Chapter 8, *LabVIEW VISA Tutorial*.

Testing Communication with Your Instrument

If you are having difficulty communicating with your instrument using the instrument driver VIs, use the Easy VISA IO VIs to test simple reads and writes interactively without running the VISA Open and VISA Close VIs. For example, for the Easy VISA Write VI, you only need to provide the resource name and the message to be sent to the instrument. The Easy VISA IO VIs include the following and are shown in the following illustration:

- Easy VISA Find Resources
- Easy VISA Write
- Easy VISA Read
- Easy VISA Write & Read
- Easy VISA Serial Write & Read
- Easy Register Write
- Easy Register Read



Although these Easy VISA IO VIs are convenient for testing purposes, they should be used with caution for application development. For each read or write to the instrument, these VIs always open and close a VISA Session, which can slow down your application if called repeatedly. For this reason, it is best to use the standard VISA functions for application development.

Developing a Quick and Simple LabVIEW Instrument Driver

Although National Instruments continues to develop new instrument drivers, we do not have the resources to develop drivers for all the requested instruments. You might find yourself in a situation where you need to interface with an instrument and no driver is available. This section describes how to develop a simple instrument driver for your application.

Modifying an Existing Driver

Before you start from scratch, check that no driver exists for your instrument. This might include checking both the manufacturer's web site, as well as National Instruments' web site. While you are checking the web sites, you should be on the lookout for instrument drivers that support a similar instrument. Instruments from the same model series often have similar command sets. Similarly, SCPI instruments of like functionality also have similar command sets. Obtain these drivers and assess the command set similarity to your instrument. For instruments from the same model series, you might need to contact the manufacturer and ask for details on the differences between the command sets. If you are comparing similar SCPI instruments, you will need to compare the instrument driver

commands with those in your instrument's programming manual.
You might want to modify an existing driver to optimize the code. For the driver to be used by a variety of users, a component VI might attempt to change a setting that is not necessary for your application. In general, you will only want to optimize those VIs that are called repeatedly in a loop. Configuration VIs are generally only called once, and have little effect on application speed.

The simplest way to modify an instrument driver VI is to rename it by selecting **Save As...** from the **File** menu. To identify the new VI you should change the name by either modifying the prefix or the description. For example, if you were modifying a Tektronix TDS oscilloscope instrument driver to work with a different instrument, you might want to rename the VI prefixes from TKTDS7XX to a name appropriate for your instrument. Once you have modified the name, you will want to modify the block diagram and front panel controls. Most changes to the block diagram will be related to the string functions. If you are not familiar with the string functions, such as Pick Line and Append or Select and Append, refer to the LabVIEW *Online Reference* for more information.

Each Initialize VI optionally calls an Identification Query that is specific to an instrument model or model series. You will either need to turn off this option, or you will need to change the response to the identification query command. For SCPI instruments, this command is *IDN?.

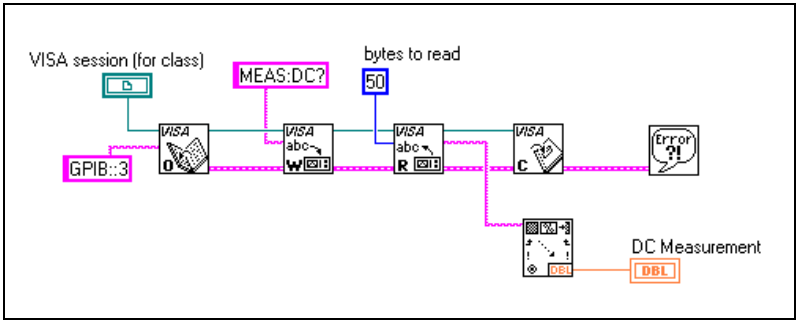
The degree to which an instrument driver needs to be changed will depend on how similar the instruments and their command sets are. If the command sets are very different, you will be better off starting from scratch.

Developing a Simple Driver

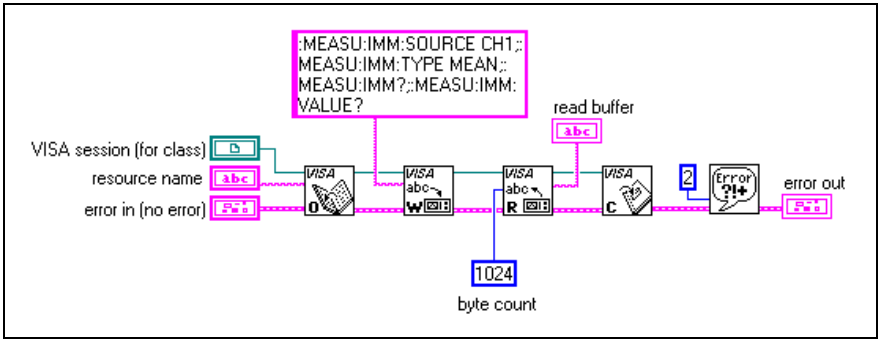
Most message-based instruments are controlled programmatically by a series of writes and reads to and from the instrument. For most simple instrument drivers only four VISA functions are needed: VISA Open, VISA Write, VISA Read, and VISA Close.

The simple instrument driver VI shown in the following illustration makes just one write and one read from an instrument. This VI first opens resources to the instrument using the VISA Open VI. Then, it sends the MEAS:DC? command (as described in the instruments user manual) to return a DC measurement from the instrument. The VISA Read function returns the measurement in the form of a string. To use the measurement with other numeric functions, the string is converted to a numeric using the Scan from String function. After completing the last read or write to the instrument, the VISA Close function is called. This is followed by the

Simple Error Handler VI to process any errors that might have occurred with the Instrument IO functions.

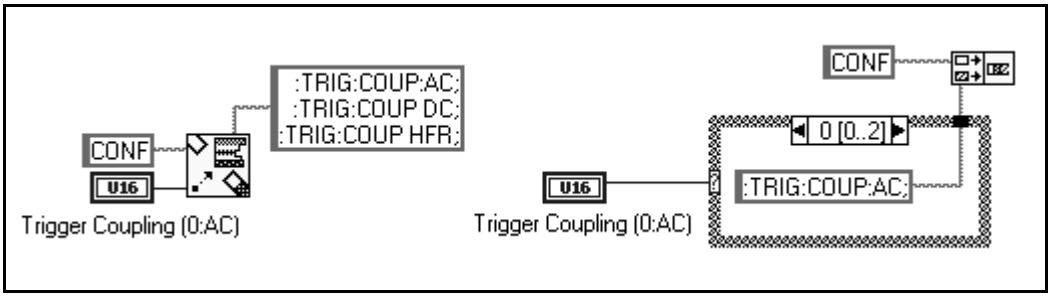


For a more modular instrument driver, you might try to break up the reads and writes to the instrument according to the type of operation you are trying to accomplish. You might want to combine the reads and writes necessary to set up configuration into one VI, while measurement reading is in another VI. For repeated measurements, you could place your measurement VIs in a loop. If you know exactly what the configuration of your setup is, you probably could include all the configuration commands into one string constant, as shown below.



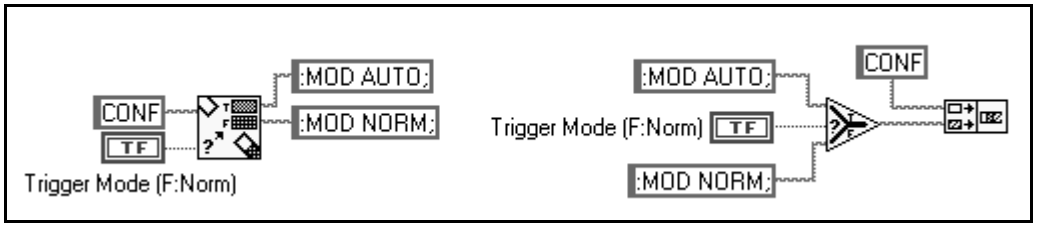
If, on the other hand, you want a user to be able to select different configurations, then you will need to programmatically build command strings. You can use the Pick Line & Append function to choose from a selection of strings and concatenate it to another string in a single step. This procedure is easier than using a Case structure and the Concatenate Strings function.

The block diagram on the left in the following illustration is easier than the one on the right.



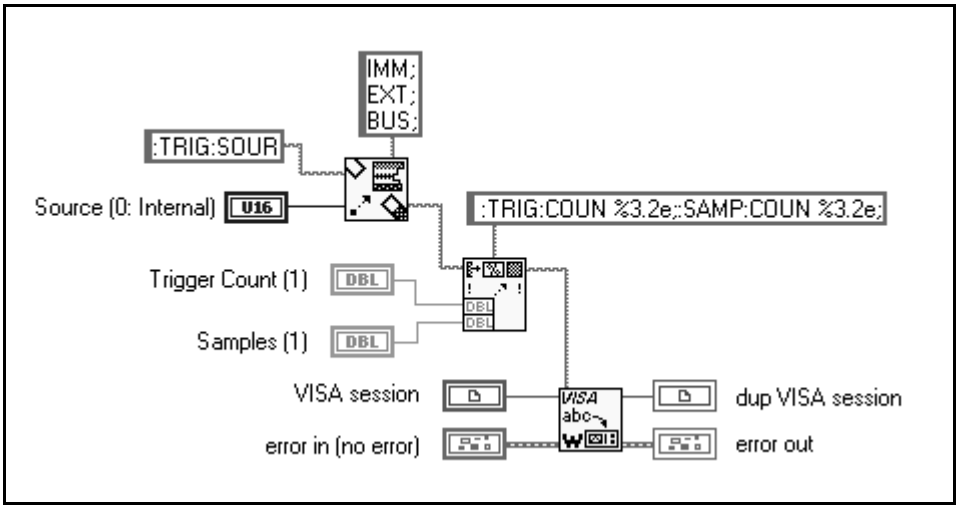
By using the Select & Append function you can select a string constant and concatenate it to another string in a single step. This procedure is easier than using a Select function followed by a Concatenate function.

The block diagram on the left in the following illustration is easier than the one on the right.

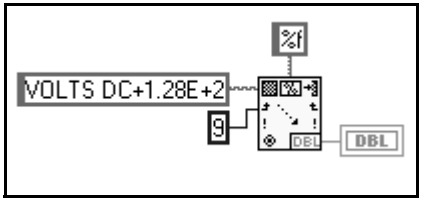


Other useful string functions for building command strings are the Format into String function and the Format & Append function. These functions convert one or more numerics into a string with a variety of formatting

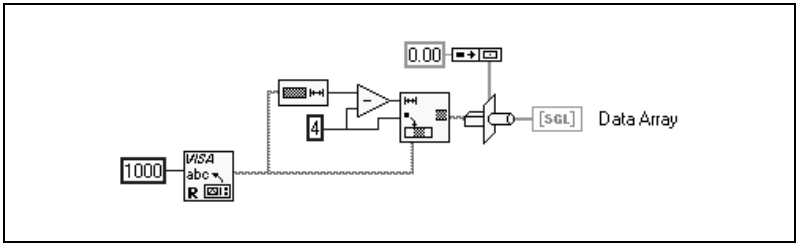
options. Both the Pick Line & Append and the Format into String functions are demonstrated in the block diagram shown below:



After you read the response from the instrument, you should parse the measurement into a numeric value. The Scan from String function is useful for converting ASCII numbers to numerics. The following code strips off the “VOLTS DC” part of the string input and converts the “+1.28E+2” to a double precision numeric. The string input is typical of a response from a multimeter.



If your instrument returns binary data, use the Type Cast function. This function changes the data type of a wire, but not how the data is stored in memory. VISA Reads return string data, regardless of whether it is encoded as ASCII or binary. Therefore, to convert the binary string to a numeric or numeric array, you need to type case the string to a different datatype. The following example strips off a 4-byte header from a 1,000-byte response string and then converts the remaining values to a single precision array.



Developing a Full-Featured Driver

If you are developing a driver that will be used by others, you might want to consider developing a full-featured driver. These drivers are more modular and have an architecture similar to those in the National Instruments Instrument Driver Library, complete with error reporting and utility functions. For more details about developing a more detailed driver, refer to the Application Note 006, *Developing a LabVIEW Instrument Driver,* on the National Instruments web site.

Using LabVIEW with IVI Instrument Drivers

In addition to the more than 600 LabVIEW source code drivers, you can control instruments with IVI (Intelligent Virtual Instruments) instrument drivers. IVI instrument drivers are DLL-based drivers developed in LabWindows/CVI that give production test users additional benefits, including the following.

- Instrument state caching for improved performance
- Simulation
- Multithread safety
- Instrument attribute access

See the LabVIEW *Online Reference* for more information about IVI instrument drivers and how to use them.

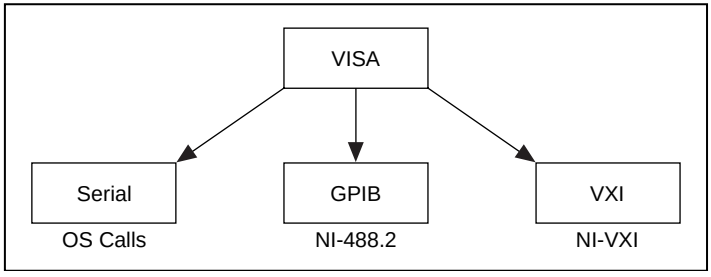
8

LabVIEW VISA Tutorial

This chapter is an overview of LabVIEW's implementation of VISA. It explains the basic concepts involved in programming instruments with VISA and gives examples demonstrating simple VISA concepts.

What is VISA?

VISA is a standard I/O Application Programming Interface (API) for instrumentation programming. VISA by itself does not provide instrumentation programming capability. VISA is a high-level API that calls into lower level drivers. The hierarchy of NI-VISA is shown in the figure below.



VISA can control VXI, GPIB, or serial instruments, making the appropriate driver calls depending on the type of instrument being used. When debugging VISA problems it is important to keep in mind that this hierarchy exists. What may appear to be a VISA problem could in reality be a problem with one of the drivers into which VISA is calling.

Supported Platforms and Environments

Because VISA is the industry standard for developing instrument drivers, most instrument drivers currently written by National Instruments use VISA and therefore support Macintosh, Windows 3.*x*, Windows 95, Windows NT, Solaris 1, Solaris 2, and HP-UX, if the system-level drivers are available for that platform.

Why Use VISA?

VISA Is the Standard

VISA is the standard API for instrument drivers throughout the instrumentation industry. In addition, you can use one API to control a suite of instruments of different types, including VXI, GPIB and serial.

Interface Independence

VISA uses the same operations to communicate with instruments regardless of the interface type. For example, the VISA command to write an ASCII string to a message-based instrument is the same whether the instrument is serial, GPIB, or VXI. Thus VISA provides interface independence. This makes it easier to switch bus interfaces, which means that users who must program instruments for different interfaces only need to learn one API.

Platform Independence

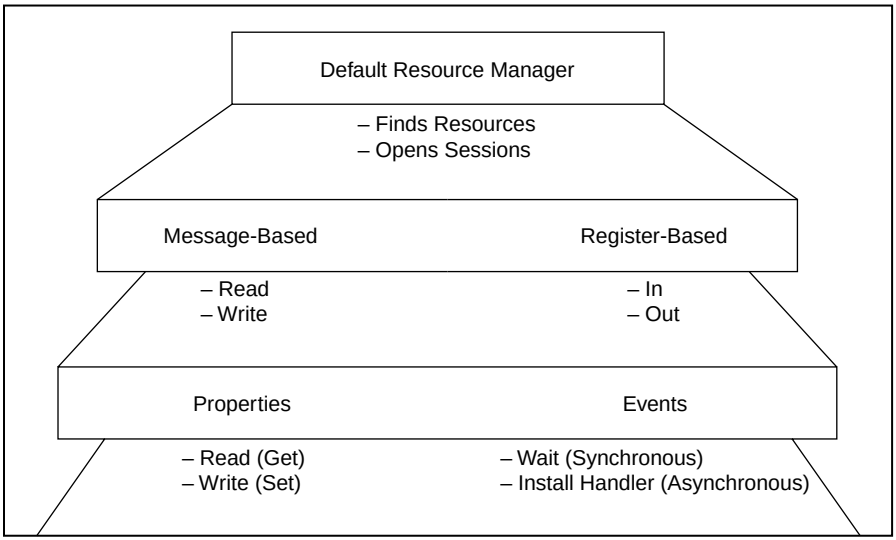
VISA is designed so that programs written using VISA function calls are easily portable from one platform to another. To ensure platform independence, VISA strictly defines its own data types. Therefore issues like the size, in bytes, of an integer variable from one platform to another should not affect a VISA program. The VISA function calls and their associated parameters are uniform across all platforms. Software can be ported to other platforms and then recompiled. A LabVIEW program can be ported to any platform supporting LabVIEW.

Easily Adapted to the Future

Another advantage of VISA is that it is an object-oriented API that will easily adapt to new instrumentation interfaces as they are developed in the future, making application migration to the new interfaces easy.

Basic VISA Concepts

A simplified outline of the internal structure of the VISA API is shown in the diagram below.



Default Resource Manager, Session, and Instrument Descriptors

The Default Resource Manager is at the highest level of VISA operations. LabVIEW automatically establishes communication with the default Resource Manager at the first VISA VI call. This brings up two terms that need to be defined: resource and session.

Resource—An entity with which you can communicate. Examples include instrument (INSTR) and memory access (MEMACC) resources.

Session—A connection (link) to any existing resource, including the Default Resource Manager.

You use the VISA Default Resource Manager to open sessions to other resources. You must open sessions to the instruments before a VI can communicate with them.

The VISA Default Resource Manager can also search for available resources in the system. You can then open sessions to any of these resources.

How Do I Search for Resources?

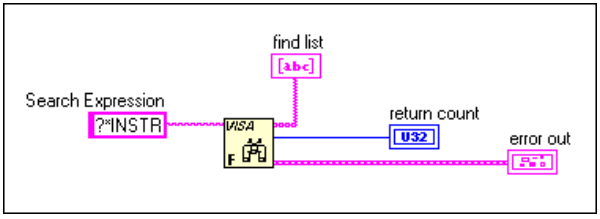
The VISA Find Resource function shown below searches for available resources in the system. This function is a common starting point for a VISA program. You can use it to determine if all of the necessary resources for the application to run are available.

The only necessary input to the VISA Find Resource function is a string called the search **expression**. This determines what types of resources the Find Resource VI will return. Possible strings for the search **expression** are shown in the table below and can be found in the LabVIEW *Online Reference*.

Instrument Resources	Expression
GPIB	GPIB[0 – 9]*::? *INSTR
GPIB-VXI	GPIB-VXI?*INSTR
GPIB or GPIB-VXI	GPIB?*INSTR
VXI	VXI?*INSTR
All VXI	?*VXI[0 – 9]*::?*INSTR
serial	ASRL[0 – 9]*::?*INSTR
All	?*INSTR

Memory Access Resources	Expression
VXI	VXI?*MEMACC
GPIB-VXI	GPIB-VXI?*MEMACC
All VXI	?*VXI[0–9]*::?*MEMACC
All	?*MEMACC

The return values of the function are the **return count** (which reports the number of resources that were found) and the **find list**. The **find list** is an array of strings. Each string contains the description of one of the resources that was found. These strings are known as instrument descriptors. A VI that will find all of the available resources in the system is shown in the figure below.



Instrument Descriptor—The exact name and location of a VISA resource. This string has the following format:

Interface Type[board index]::Address::VISA Class

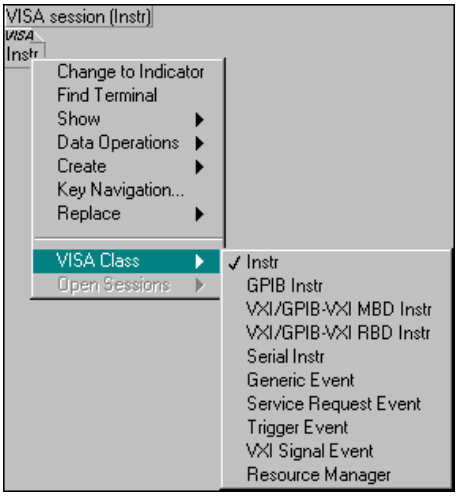
The instrument descriptors are simply specific instruments found by the search query. The board index only needs to be used if more than one interface type is present in the system. For example, if the system contains two GPIB plug in boards one could be referred to as GPIB0 and one as GPIB1. In this case, the board index needs to be used in instrument descriptors. For VXI instruments the Address parameter is the Logical Address of the instrument. For GPIB instruments it is the GPIB primary address. Serial instruments do not use the address parameter. For example, ASRL1::INSTR is the descriptor for the COM 1 serial port on a personal computer.

What is a VISA Class?

The VISA Class is a grouping that encapsulates some or all of the VISA operations. INSTR is the most general class that encompasses all VISA operations for an instrument. In the future other classes might be added to the VISA specification. Currently the VISA Class does not need to be included as part of the instrument descriptor, but you should include it to ensure future compatibility. Currently, if the VISA class is left blank, it will default to the INSTR class.

Popping Up on a VISA Control

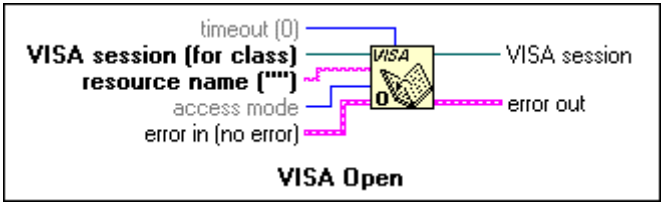
LabVIEW supplies another way to set the class for a VISA session that can be used now. You can pop up on the front panel VISA Session control and select the VISA Class as shown in the following figure.



If a class other than the default Instr class is selected, only functions for operations associated with that device class can be wired successfully with this session. For example, if GPIB Instr is selected for the VISA class, the functions for VXI register access can not be wired with the session.

Opening a Session

The instrument descriptors are used to open sessions to the resources in the system. The VISA Open function is shown below.

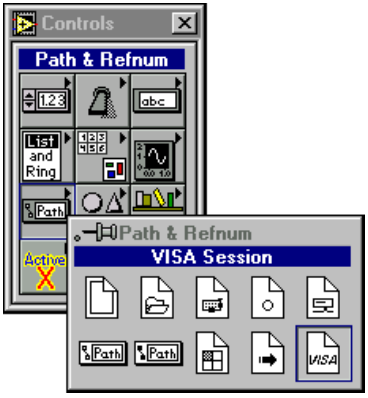


The resource name input is the VISA instrument descriptor for the resource to which a session will be opened.

Note ***You do not need to use the Find Resource VI to obtain instrument descriptors for resources. The VISA Open VI can be used with an instrument descriptor provided by the user or programmer. However, to be sure of the syntax for the descriptor it is best to run Find Resource first.***

Note ***In most applications, sessions only need to be opened once for each instrument that the application will communicate with. This session can be routed throughout the application and then closed at the end of the application.***

Notice that there is also a VISA session input to the VISA Open VI. To open sessions to resources in LabVIEW a VISA Session front panel control is needed. The VISA session front panel control can be found in the **Controls** Palette in the **Path & Refnum** subpalette.



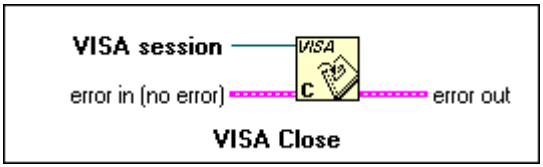
How Do the Default Resource Manager, Instrument Descriptors, and Sessions Relate?

It is important to have a clear understanding at this point of the Default Resource Manager, instrument descriptors, and sessions. A good analogy can be made between the VISA Default Resource Manager and a telephone operator. Opening a session to the Default Resource Manager (remember this is done automatically in LabVIEW) is like picking up the phone and calling the operator to establish a line of communication between a program and the VISA driver.

In turn the telephone operator can dial phone numbers to establish lines of communication with resources in the system. The phone numbers that the resource manager uses are the instrument descriptors. The lines of communication are the sessions that are opened to VISA resources. In addition, the resource manager can look for all the available phone numbers. This is the VISA Find Resource operation.

Closing a Session

An open session to a VISA resource also uses system resources within the computer. To properly end a VISA program, all of the opened VISA sessions should be closed. To do this there is a VISA Close VI that is shown below.



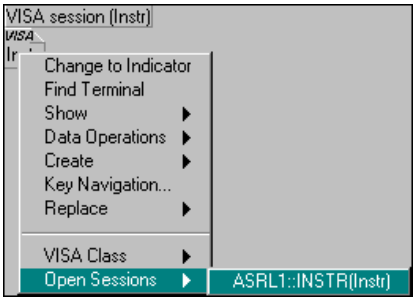
The VISA session input to the VISA Close VI is the session to be closed. This session originally comes from the output session terminal of the VISA Open VI.

If a session is not closed when a VI is run, it remains open. It is a good idea to close sessions that are opened in an application so open sessions don't build up and cause problems with system resources. However, there are cases when leaving sessions open can be useful.

☞ **Note** ***If a VI is aborted (for example, when you are debugging a VI) the VISA session is not closed automatically. You can use the Open VISA Session Monitor VI (located in*** `vi.lib/Utility` ***) to assist in closing such sessions.***

When Is It a Good Idea to Leave a Session Open?

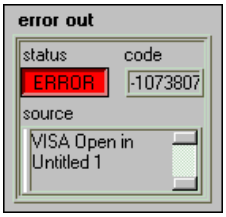
If a VI runs and leaves a session open without closing it, this session can be used in later VIs. An open session can be accessed by popping up on a VISA Session front panel control and choosing **Open Sessions**. The output of the VISA Session front panel control will then be the selected open session. In this way sessions can be closed that were left open from earlier runs of a VI. This method can also be used to interactively test parts of an application. An example of selecting an open session is shown in the following figure.



You can use the VISA Session control to check for open sessions. Accessing open sessions by popping up on front panel VISA Session Controls also provides a convenient way to interactively run parts of an instrument driver.

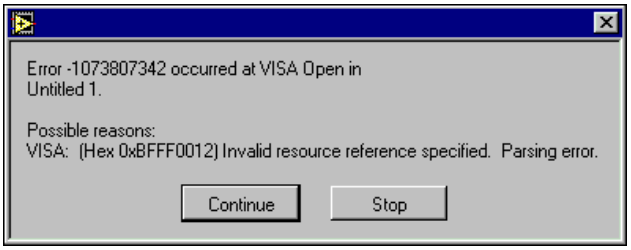
Error Handling with VISA

Error handling with VISA VIs is similar to error handling with other I/O VIs in LabVIEW. Each of the VISA VIs contain Error Input and Error Output terminals that are used to pass error clusters from one VI to another in a diagram. The error cluster contains a Boolean flag indicating whether an error has occurred, a numeric VISA error code, and a string containing the location of the VI where the error occurred. If an error occurs subsequent VIs will not try to execute and simply pass on the error cluster. A front panel error cluster indicator showing the output from the error out terminal of a VISA VI is shown in the figure below.

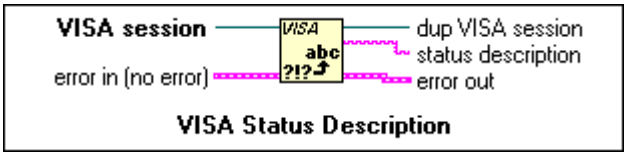


Notice that in this case an error has occurred. Also notice that the VISA error code is cut off in the code indicator. VISA error codes are 32-bit integers that are usually in hexadecimal format. The LabVIEW error cluster displays the code in decimal. The *VISA Error Codes* topic in the LabVIEW *Online Reference*, and the *Numeric Error Codes* section in Appendix A, *Error Codes*, of the *LabVIEW Function and VI Reference Manual*, also list the error codes in decimal. However, as the figure above shows, these error codes are cut off in the error cluster. The code indicator must be resized to display the entire error code.

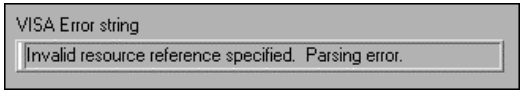
The LabVIEW Simple and General Error Handler VIs can be found in the **Time & Dialog** subpalette under the **Functions** palette. If an error occurs, these VIs provide a pop-up dialog box, which displays possible reasons for the error. The Simple Error Handler VI returns the same error as the error cluster used in the previous example, but provides more detailed information about the error, as shown in the following figure.



Notice that the code description is listed under the possible reasons. It is not always convenient to handle errors with pop up dialog boxes through the LabVIEW error handling VIs. VISA also provides an operation that will take a VISA error code and produce the error message string corresponding to the code as an output. This VI is shown in the figure below.



The inputs to this VI are a VISA session and a VISA error cluster. The VI will check the VISA code in the input error cluster and output the text description of the code in **Status Description**. The following figure shows a LabVIEW string indicator displaying the error string returned from the VISA Status Description VI.



The exact method used for implementing error handling will depend on the nature of the program. However, some sort of error handling mechanism should be implemented in any program involving VISA.

Easy VISA VIs

You can use the Easy VISA VIs to verify that you have established communication with your instrument. When developing your applications, you should use the other VISA VIs in the palette because they give you more control over your instrument. For more information about the Easy VISA VIs, see the *Testing Communication with Your Instrument* section of Chapter 7, *Getting Started with a LabVIEW Instrument Driver*. The examples in the following sections do not use the Easy VISA VIs.

Message-Based Communication

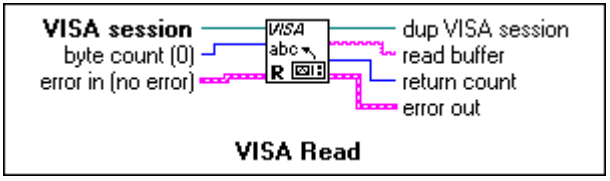
Serial, GPIB, and many VXI devices recognize a variety of message-based command strings. At the VISA level the actual protocol used to send a command string to an instrument is transparent. A user only needs to know that they would like to write a message to or read a message from a message-based device. The VIs that are used to perform these operations are VISA Write and VISA Read.

Note ***These same VIs are used to write message-based commands to GPIB, serial, and message-based VXI instruments. VISA knows automatically which driver functions to call based on the type of resource being used.***

The VISA Write VI is shown below.

The only input, besides the session, is the string to be sent to the instrument.

The VISA Read VI is equally easy to use. It is shown below.



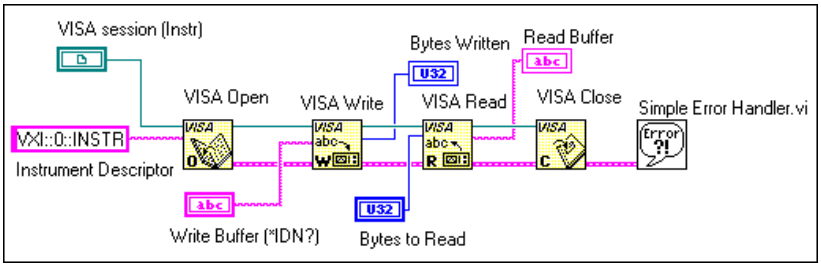
The VISA Read VI must be given a byte count input equal to the maximum number of bytes that should be read from the instrument. The VI will stop

reading when this number of bytes has been read or when the end of the transfer is indicated.

The actual message-based commands that the instrument recognizes vary from manufacturer to manufacturer. IEEE 488.2 and SCPI standardized the commands for message-based instruments. Many instruments follow these standards. However, the only way to be certain of the commands for a particular instrument is to refer to the documentation provided by the manufacturer. However, instrument drivers exist for many message-based devices. These instrument drivers contain functions that put together the appropriate ASCII command strings and send them to the instrument. See the National Instruments website or ftp site to obtain the latest drivers.

How Do I Write To and Read From a Message-Based Device?

A simple example that writes the *IDN? (identification) string to a message-based instrument and reads the response is shown in the figure below.



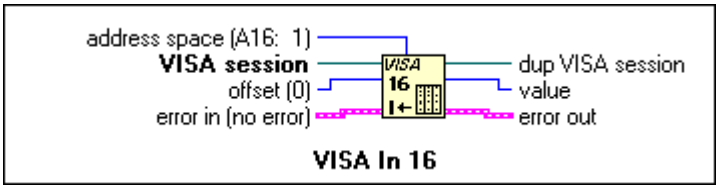
This program could be used successfully with any device that recognizes the *IDN? command. The device could be serial, GPIB, or message-based VXI. The only change would be the instrument descriptor.

Register-Based Communication (VXI only)

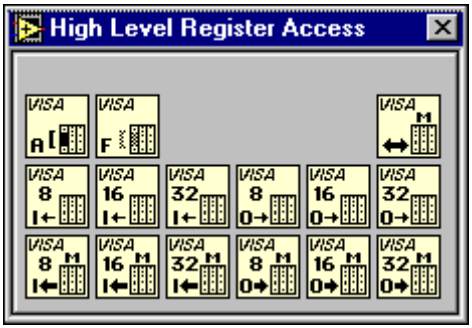
VISA contains a set of register access VIs for use with VXI instruments. If you are using GPIB or serial devices only, this section does not apply.

Some VXI instruments do not support message-based commands. The only way to communicate with these instruments is through register accesses. All VXI instruments have configuration registers in the upper 16 kilobytes of A16 memory space. Therefore, register access functions can also be used to read from and write to the configuration registers for message-based devices. The basic VISA operation used to read a value from a register is VISA In. There are actually three different versions of this operation for

reading an 8-, 16-, or 32-bit value. You must use the VI for the access width that your instrument supports. For example, a particular instrument might return a bus error for 32-bit accesses if it were designed for 16-bit access. The VISA In 16 VI is shown in the following figure.



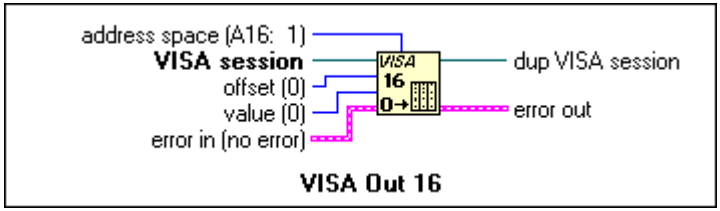
This VI and the other basic register access VIs can be found in the **High Level Register Access** subpalette under the main VISA function palette.



The address space input indicates which VXI address space to use. The offset input sometimes causes confusion. Remember that VISA keeps track of the base memory address that a device requests in each address space. The offset input is relative to this base address.

Consider the following example. Suppose that you have a device at Logical Address 1 and would like to use the VISA In 16 VI to read its ID/Logical Address configuration register. You know that this register is at absolute address 0xC040 in A16 space and that the configuration registers for the device at Logical Address 1 lie from 0xC040 to 0xC07F. However, VISA also knows this and you only need to specify the offset in this region you wish to access. In this case that offset is zero.

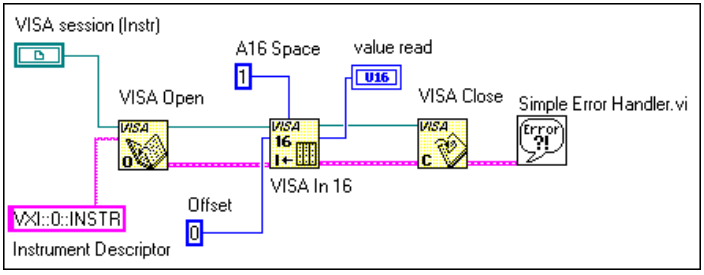
There is another set of high-level register access operations that parallel the VISA In operations, but are for writing to registers. These operations are the VISA Out operations. The VISA Out 16 VI is shown below.



This VI is similar to the VISA In 16 VI except the value to write must be provided to the value terminal. Keep in mind when using the VISA Out VIs that some registers may not respond to a write cycle or may cause a bus error.

Basic Register Access

An example of using the high-level VISA access functions in a VI is shown in the simple program below.

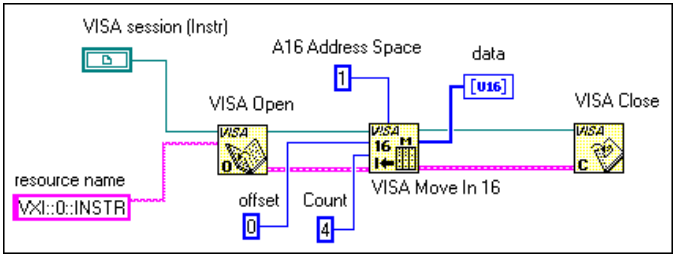


This block diagram shows how to use the VISA In 16 VI to read the first configuration register for a VXI device at Logical Address 0. The offset parameter, in this case zero, is relative to the memory range that the device requests in the VXI address space being accessed. The address space parameter indicates which VXI address space is being accessed. In this case the device is at Logical Address 0. Its configuration registers are located from 0xC000 to 0xC03F in the A16 address space. The VISA In 16 operation with offset 0 is really reading the register at 0xC000.

The program reads from a 16-bit register in the A16 address space at the specified offset (0) of the specified resource (VXI::0::INSTR). If an error occurs in the sequence of VISA VIs that execute in the program diagram, the Simple Error Handler returns a dialog box informing the user of the error and displaying the text message associated with the VISA Error Code.

Basic Register Move

The following block diagram shows how to use the VISA Move In 16 VI to read the first four configuration registers for a VXI device at Logical Address 0.



The VISA Move In VIs are used to read large blocks of data from VXI devices. The data is returned as an array of four 16-bit values. There is a corresponding set of VISA Move Out VIs for moving large blocks of data to VXI devices. The Move In and Move Out VIs have 8-, 16-, and 32-bit versions. The appropriate VI is determined by the size of the registers that are going to be accessed.

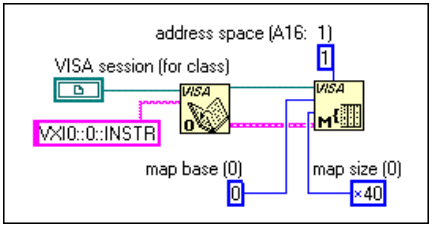
Low-Level Access Functions

Low-Level Access (LLA) functions provide a very efficient way to perform register-based communication. LLA functions incur much less overhead than High-Level Access (HLA) functions for certain types of accesses. LLA functions perform the same steps that the HLA functions do, except that each individual task performed by an HLA function is an individual function under LLA.

Using VISA to Perform Low-Level Register Accesses

The first LLA operation you need to call for accessing a device register is the `VISA Map Address` operation, which sets up the hardware window to allow access to the VXI address space. The `VISA Map Address` operation programs the hardware to map local CPU addresses to VXI

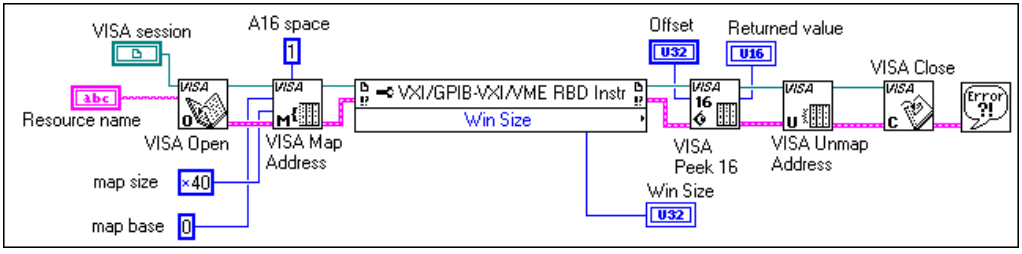
addresses as described in the previous section. The following code is an example of programming the hardware to access A16 space.



This sample code sets up the hardware to map A16 space, starting at offset 0 for 0x40 bytes. Remember that the offset is relative to the base address of the device we are talking to through the `VISA` session, not from the base of A16 space itself. Therefore, offset 0 does not mean address 0 in A16 space, but rather the starting point of the device's A16 memory.

Note ***To access the device registers through a MEMACC session, you need to provide the absolute VXIbus addresses (base address for device + register offset in device address space).***

If you need more than a single map for a device, you will need to open a second session to the device because VISA currently supports only a single map per session. There is very low overhead in having two sessions because sessions themselves do not take much memory. However, you will need to keep track of two session handles. Notice that this is different from the maximum number of windows you can have on a system. The hardware for the controller you are using may have a limit on the number of unique windows it can support. When you are finished with the window, or you need to change the mapping to another address or address space, you must first unmap the window using the `VISA Unmap Address` operation.



This example maps 64 bytes (hex 40) to the A16 address space starting at offset 0 from the address of the device at Logical Address 1. When using LLA operations, always specify a map size that is large enough to

accommodate the range of addresses you will access. You can do this by using a property node to determine the amount of address space used by the device. Property nodes are explained later in this chapter.

☞ **Note** ***The default maximum window that can be mapped is typically 64kB. If using a MITE-based controller, you can request more than 64kB, but you will need to increase your User Window size. This is done in the resource editor for your controller, either T&M Explorer, VXIEdit, or VXItedit. Please consult the documentation that came with your controller.***

Bus Errors

Bus errors are not reported by the LLA operations. In fact, `VISA Peek` and `VISA Poke` do not report any error conditions. However, the HLA operations do report bus errors. When using the LLA operations, you must ensure that the addresses you are accessing are valid.

Comparison of High-Level and Low-Level Access

Speed

In terms of the speed of developing your application, the HLA operations are much faster to implement and debug because of the simpler interface and the status information received after each access. For example, HLA operations encapsulate the mapping and unmapping of hardware windows, which means that you do not need to call `VISA Map Address` and `VISA Unmap Address` separately.

For speed of execution, the LLA operations perform faster when used for several random register I/O accesses in a single mapped window. If you know that the next several accesses are within a single window, you can perform the mapping just once and then each of the access has minimal overhead.

The HLA operations will be slower because they must perform a map, access, and unmap within each call. Even if the window is correctly mapped for the access, the HLA call at the very least must perform some sort of check to determine if it needs to remap. Also, because HLA operations encapsulate many status checking capabilities not included in LLA operations, HLA operations have higher software overhead. For these reasons, HLA is slower than LLA in many cases.

☞ **Note** ***For block transfers, the high-level VISA Move operations perform the fastest.***

Ease of Use

HLA operations are easier to use because they encapsulate many status checking capabilities not included in LLA operations, which explains the higher software overhead and lower execution speed of HLA operations. HLA operations also encapsulate the mapping and unmapping of hardware windows, which means that you do not need to call `VISA Map Address` and `VISA Unmap Address` separately.

Accessing Multiple Address Spaces

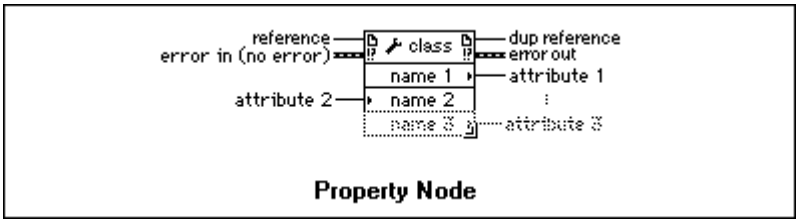
You can use LLA operations to access only the address space currently mapped with a single VISA session. For a single session, to access a different address space, you need to perform a remapping, which involves calling `VISA Unmap Address` and `VISA Map Address`. Therefore, LLA programming becomes more complex for accessing several address spaces concurrently.

In addition, if you have several sessions to the same or different devices all performing register I/O, they must compete for the finite number of hardware windows available. When using LLA operations, you must allocate the windows and always ensure that the program does not ask for more windows than are available. The HLA operations avoid this problem by restoring the window to the previous setting when they are done. Even if all windows are currently in use by LLA operations, you can still use HLA functions because they will save the state of the window, remap, access, and then restore the window. As a result, you are not restricted when using the HLA operations.

VISA Properties

The basic operations that are associated with message and register-based resources in VISA have now been introduced. These operations allow register access, and message-based communication. In addition to the basic communication operations, VISA resources have a variety of properties (attributes) with values that can be read or set in a program.

In a LabVIEW program these properties are handled programmatically in the same way that the properties of front panel controls and indicators are handled. Property nodes are used to read or set the values of VISA properties. The property node is shown in the following figure.



☞ **Note** ***The property node is a generic node that also can be used to set ActiveX/OLE and VI Server properties.***

After placing a property node on the block diagram, you can set the properties for a VISA class. You can use either of the following methods to do this.

- Wire a VISA Session to the reference input terminal of the property node.
- Pop up on the property node and choose **Instr** from the **Select VISA Class** menu.

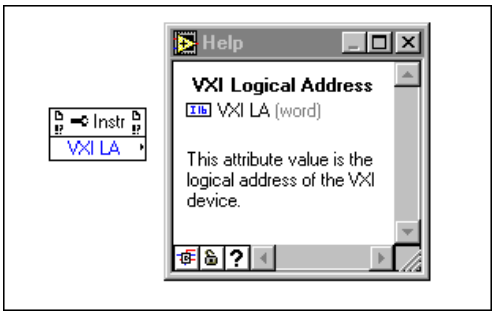
The property node contains a single property terminal when it is initially placed on the block diagram. However, it can be resized to contain as many terminals as necessary. The initial terminal on the VISA property node is a read terminal. This means that the value of the property selected in that terminal will be read. This is indicated by the small arrow pointing to the right at the right edge of the terminal. Many terminals can be changed individually from a read terminal to a write terminal by popping-up on the property you wish to change.

☞ **Note** ***Some properties are read only. Their values can not be set.***

To select the property in each terminal of the property node, pop up on the property node terminal and choose **Select Item**. This will provide a list of all the possible properties that can be set in the program. The number of different properties that are shown under the Select Item choice of the VISA Property Node can be limited by changing the VISA Class of the property node.

To change the VISA class pop up on the VISA property node and select **VISA Class**. Several different classes can be selected under this option besides the default INSTR class which encompasses all possible VISA properties. These classes limit the properties displayed to those related to that selected class instead of all the VISA properties. Once a session is connected to the **Session** input terminal of the property node, the VISA Class will be set to the class associated with that session.

Initially, the VISA properties will be somewhat unfamiliar and their exact nature may not be clear from the name alone. The LabVIEW *Online Reference* contains information on the properties. Brief descriptions of individual properties are also available in the simple help window. To get a brief description of a specific property, select the property in one of the terminals of a property node and then open the help window. This is shown for the VXI LA property below.



Note that the help window shows the specific variable type of the property and gives a brief description of what the property does. In cases where it is not clear what variable type to use for reading or writing a property, remember popping up on a property node and selecting **Create Constant**, **Create Control**, or **Create Indicator** will automatically select the appropriate variable type.

There are two basic types of VISA properties—global properties and local properties. Global properties are specific to a resource while local properties are specific to a session. For example, the VXI LA property is an example of a global property. It applies to all of the sessions that are open to that resource. A local property is a property that could be different for individual sessions to a specific resource. An example of a local property is the timeout value. Some of the common properties for each resource type are shown in the following lists.

Serial

Serial Baud Rate—The baud rate for the serial port.

Serial Data Bits—The number data bits used for serial transmissions.

Serial Parity—The parity used for serial transmissions.

Serial Stop Bits—The number of stop bits used for serial transmissions.

GPIB

GPIB Readdressing—Specifies if the device should be readdressed before every write operation.

GPIB Unaddressing—Specifies if the device should be unaddressed after read and write operations.

VXI

Mainframe Logical Address—The lowest logical address of a device in the same chassis with the resource.

Manufacturer Identification—The manufacturer ID number from the device's configuration registers.

Model Code—The model code of the device from the device's configuration registers.

Slot—The slot in the chassis that the device resides in.

VXI Logical Address—The logical address of the device.

VXI Memory Address Space—The VXI address space used by the resource.

VXI Memory Address Base—The base address of the memory region used by the resource.

VXI Memory Address Size—The size of memory region used by the resource.

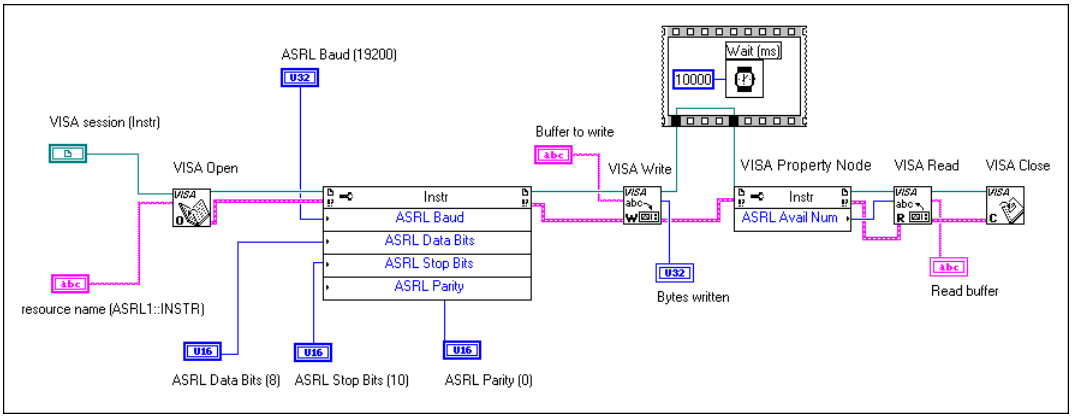
There are many other properties besides those listed here. There are also properties that are not specific to a certain interface type. The timeout property, which is the timeout used in message-based I/O operations, is a good example of such a property. The most complete source for information about properties is the LabVIEW *Online Reference*, which you can access by selecting **Help»Online Reference**.

The online help shows which type of interfaces the property applies to, if the property is local or global, its data type, and what the valid range of values are for the property. It also shows related items and gives a detailed description of the property.

VISA Property Examples

Serial Write and Read

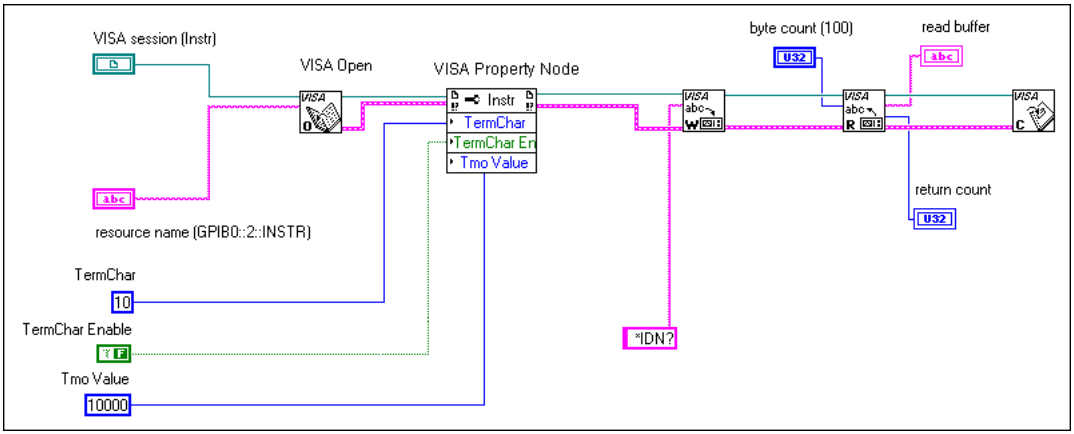
This section contains three simple examples of using properties in VISA programs. The first, shown in the following figure, writes a string to a serial instrument and reads the response.



The VI opens a session to the serial port COM 1 and initializes it for 19,200 baud, 8 data bits, no parity, and 1 stop bit. A string is then written to the port. After writing the string and waiting for 10 seconds the number of bytes that have been returned by the device are obtained using another VISA property. These bytes are then read from the port. Notice that you use the value 10 to set the number of stop bits to one. (This is from the VISA specification. 10 corresponds to 1 stop bit, 20 to 2 stop bits.)

How Do I Set a Termination Character for a Read Operation?

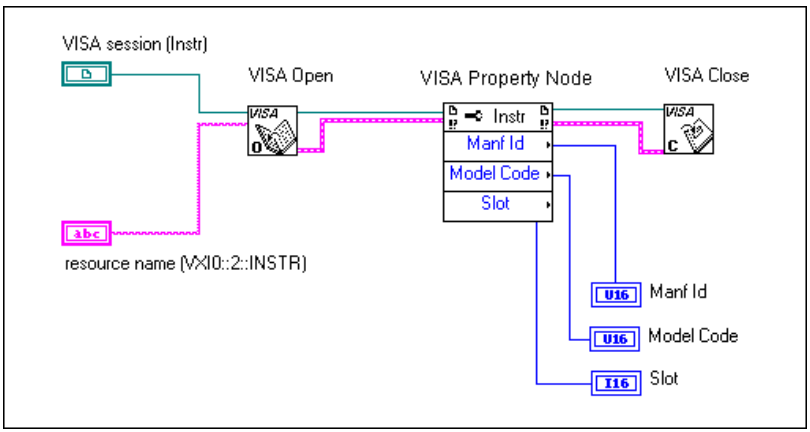
The next example shows how to use properties to set a termination character for VISA read operations. Some message-based devices send a special termination character when they have no more data to send.



This VI opens a session to GPIB instrument at primary address 2. The VI sets the termination character to a linefeed (decimal value 10) and then enables the use of a termination character with another property. The VI also sets the timeout property to 10,000 milliseconds (10 seconds). It then writes the string *IDN? to the instrument and tries to read back a 100 character response. The read will terminate when the termination character is received. The VI stops when the termination character is received, after it reads 100 bytes, or after 10 seconds.

VXI Properties

The final example shows how to read some of the common properties associated with a VXI instrument.



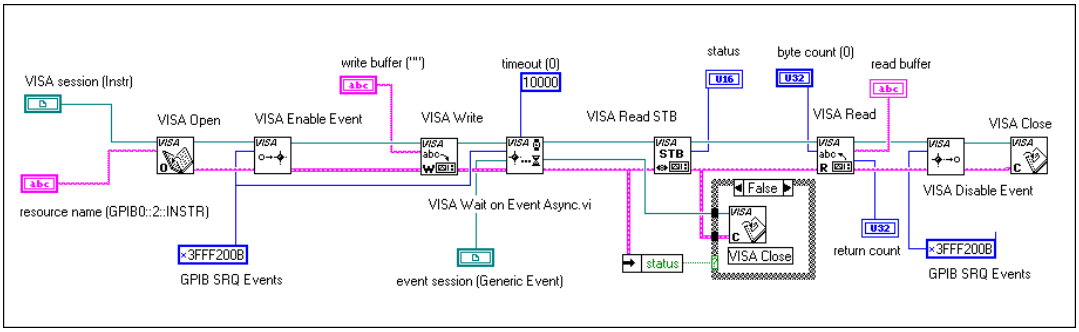
This VI opens a session to a VXI instrument at Logical Address 2 and reads the manufacturer ID, model code, and slot for the VXI module.

Events

An event is a VISA means of communication between a resource and its applications. It is a way for the resource to notify the application that some condition has occurred that requires action by the application. Examples of different events are included in the following sections.

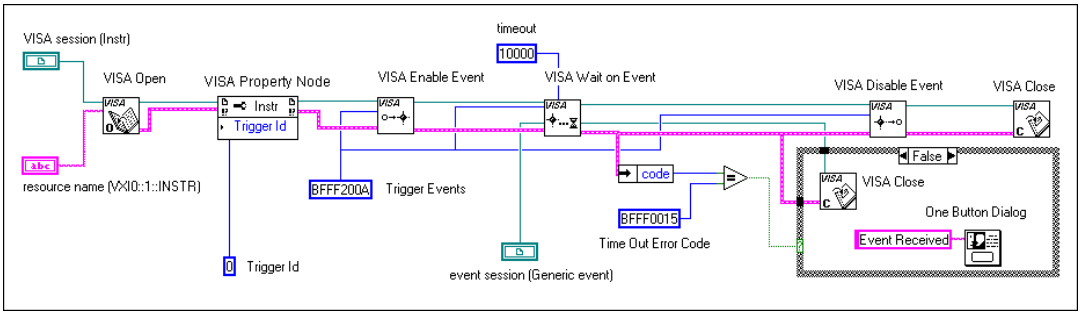
GPIB SRQ Events

The following block diagram shows how to handle GPIB Service Request (SRQ) events with VISA.



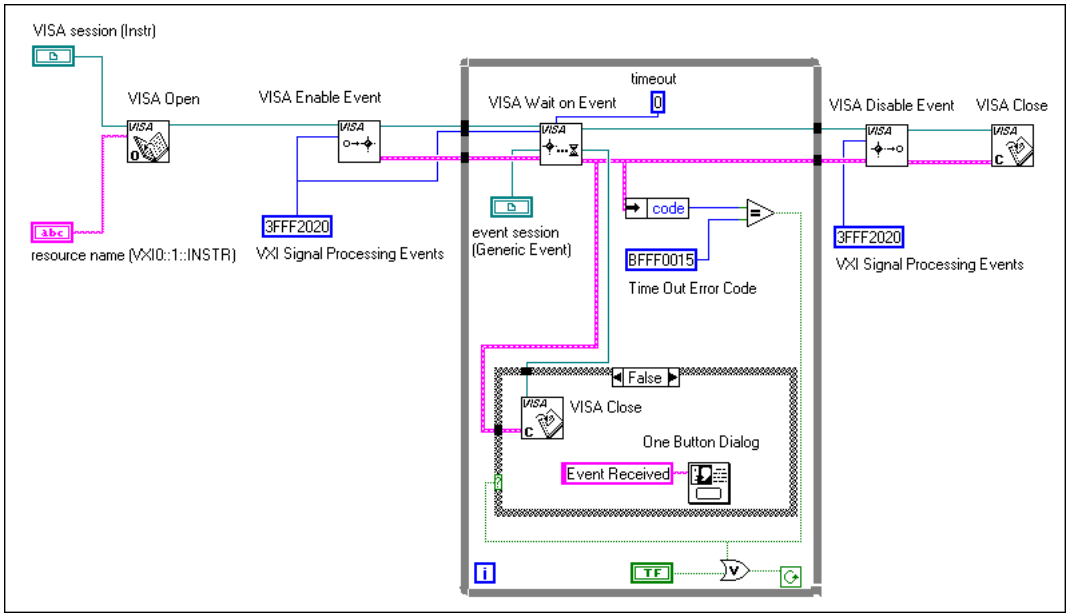
The VI enables service request events and then writes a command string to the instrument. The instrument is expected to respond with an SRQ when it has processed the string. The Wait on Event Async VI is used to wait for up to 10 seconds for the SRQ event to occur. Once the SRQ occurs, the instrument's status byte is read with the Read Status Byte VI. The status byte must be read after GPIB SRQ events occur or later SRQ events may not be received properly. Finally the response is read from the instrument and displayed. The Wait on Event Async is different from the regular Wait on Event VI in that it continuously calls Wait on Event with a timeout of zero to poll for the event. This frees up time for other parallel segments of the program to execute while waiting for the event.

Trigger Events



This diagram shows how to detect a trigger on TTL Trigger Line 0 for a device at Logical Address 1. You must set the type of trigger events to detect with a VISA property before the events are enabled. The VI waits for up to 10 seconds for the event to be received. If the event is received successfully the event is closed in the VI.

Interrupt Events

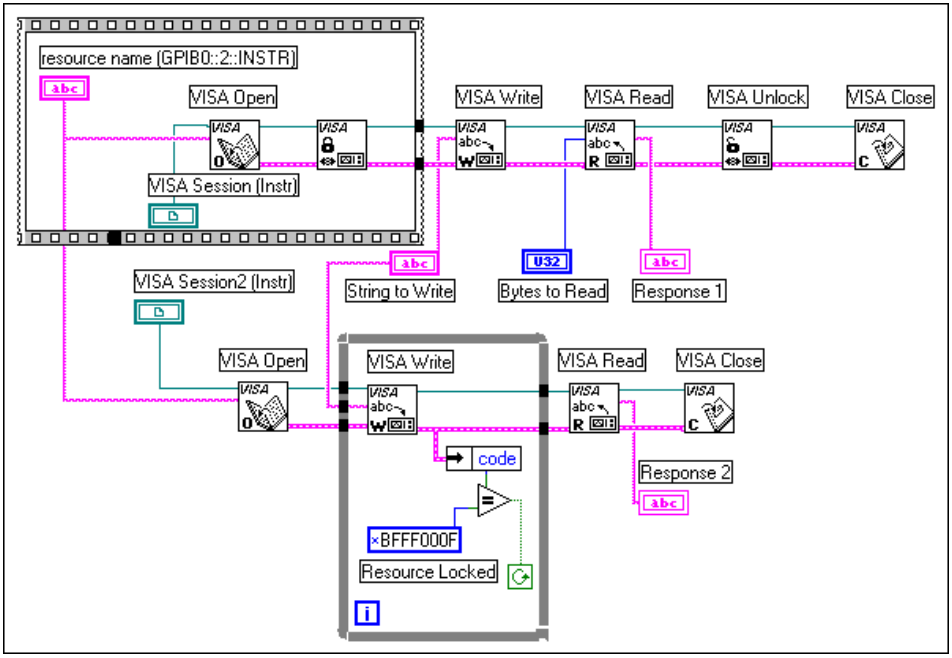


This diagram demonstrates using VISA's event handling capability to detect a VXI interrupt asserted by a VXI device at Logical Address 1. The VI enables VXI Signal Processing Events and then goes into a loop that repeatedly calls VISA Wait on Event. The loop will terminate if an event is received or if a front panel stop switch is selected. The Wait on Event VI has a timeout terminal that is set to a value of zero. In this case the VI simply checks to see if any events have been received and then immediately returns a timeout error if there is no event in the event queue. If an event is received the event session is closed and notification of the event is produced. Once the event handling is finished the events are disabled.

Locking

VISA introduces locks for access control of resources. In VISA, GPIB and VXI applications can open multiple sessions to the same resource simultaneously and can access the resource through these different sessions concurrently. In some cases, applications accessing a resource must restrict other sessions from accessing that resource. For example, an application may need to execute a write and a read operation as a single step so that no other operations take place between the write and read operations. The application can lock the resource before invoking the write operation and unlock it after the read operation, to execute them as a single step.

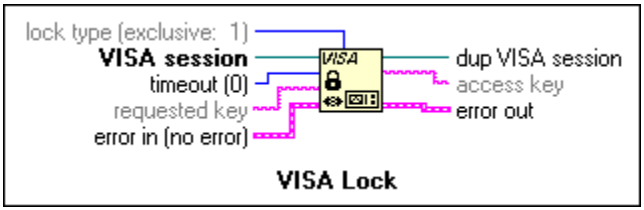
The VISA locking mechanism enforces arbitration of accesses to resources on an individual basis. If a session locks a resource, operations invoked by other sessions are serviced or returned with a locking error, depending on the operation and the type of lock used.



This VI opens two sessions to the same resource but locks the first session. The first session then writes a command to the resource and reads the response. Once the write/read sequence is completed the first session unlocks the resource. At this point the second session, which is trying to perform the same write/read, will no longer receive a resource locked error on the write operation and can complete successfully. Locking can be used in cases where more than one application may be accessing the same resource or multiple sessions will be opened to the resource in a single application.

Shared Locking

There may be cases where you want to lock access to a resource, but selectively share this access. The figure below shows the Lock VI in complex help view.



Lock type defaults to exclusive, but you can set it to shared. You can then wire a string to **requested key** to be the password needed for another application to access the resource. However, the VI assigns one in **access key** if you don't ask for one. You can then use this key to access a locked resource.

Platform-Specific Issues

This section discusses programming information for you to consider when developing applications that use the NI-VISA driver.

After installing the driver software, you can begin to develop your VISA application software. Remember that the NI-VISA driver relies on NI-488.2 and NI-VXI for driver-level I/O accesses.

- **Windows 95/NT users**—On VXI and MXI systems, use T&M Explorer to run the VXI Resource Manager, configure your hardware, and assign VME and GPIB-VXI addresses. For GPIB systems, use the system Device Manager to configure your hardware. To control instruments through serial ports, you can use T&M Explorer to change the default settings, or you can perform all the necessary configuration at run time by setting VISA attributes.
- **All other platforms**—On VXI and MXI systems, you must still run VXIinit and Resman, and use VXIedit or VXItedit for configuration purposes. Similarly, for GPIB and GPIB-VXI systems, you still use the GPIB Control Panel applet or IBCONF to configure your system. To control instruments through serial ports, you can do all necessary configuration at run time by setting VISA attributes.

Programming Considerations

This section contains information for you to consider when developing applications that use the NI-VISA I/O interface software.

Multiple Applications Using the NI-VISA Driver

Multiple-application support is an important feature in all implementations of the NI-VISA driver. You can have several applications that use NI-VISA running simultaneously. You can even have multiple instances of the same application that uses the NI-VISA driver running simultaneously, if your application is designed for this. The NI-VISA operations perform in the same manner whether you have only one application or several applications (or several instances of an application) all trying to use the NI-VISA driver.

However, you need to be careful in cases when you have multiple applications or sessions using the low-level VXIbus access functions. The memory windows used to access the VXIbus are a limited resource. You should call the viMapAddress() operation before attempting to perform low-level VXIbus access with viPeek*XX*() or viPoke*XX*(). Immediately after the accesses are completed, you should always call the viUnmapAddress() operation so that you free up the memory window for other applications.

Multiple Interface Support Issues

This section contains information about how to use and/or configure your NI-VISA software for certain types of interfaces.

VXI and GPIB Platforms

NI-VISA supports all existing National Instruments VXI, GPIB, and serial hardware for the operating systems on which NI-VISA exists. For VXI, this includes MXI-1 and MXI-2 platforms, the GPIB-VXI, and the line of VXIpc embedded computers. For GPIB, this includes, but is not limited to, the PCI-GPIB, NB-GPIB, GPIB-SPARC series, the full line of AT-GPIB/TNT boards, and the GPIB-ENET box, which you can use to remotely control GPIB devices. With the GPIB-ENET, you can even remotely control VXI devices when using a GPIB-VXI controller.

Multiple GPIB-VXI Support

Windows 95/NT users can refer to the T&M Explorer utility to add multiple National Instruments GPIB-VXI controllers, or any other vendor's GPIB-VXI controller, to your system. Windows 3.*x* and UNIX users must use the VISAconf utility to add the controllers.

Serial Port Support

NI-VISA currently supports only a single session at a time on a given serial port. The maximum number of serial ports that NI-VISA currently supports on any platform is 32. The default numbering of serial ports is system-dependent.

Platform	Method
Windows 3.*x*, Windows 95, Windows NT	`ASRL1` – `ASRL4` access `COM1` – `COM4` `ASRL10` – `ASRL13` access `LPT1` – `LPT4`
Macintosh 68K, Macintosh PPC	`ASRL1` accesses the modem port `ASRL2` accesses the printer port
Solaris 1.*x*	`ASRL1` – `ASRL6` access `/dev/ttya` – `/dev/ttyf`
Solaris 2.*x*	`ASRL1` – `ASRL6` access `/dev/cua/a` – `/dev/cua/f`
HP-UX 9 HP-UX 10	`ASRL1` and `ASRL2` access serial ports 1 and 2 through `/dev/tty00` and `/dev/tty01` on HP-UX9. HP-UX10 uses `/dev/tty0p0` and `/dev/tty1p0`. Additional ports are numbered consecutively starting at ASRL3, which uses `/dev/tty02`.

VME Support

To access VME devices in your system, you must configure NI-VXI to see these devices. Windows 95/NT users can configure NI-VXI by using the **Add Device Wizard** in the T&M Explorer. Users on other platforms must use the **Non-VXI Device Editor** in VXIedit or VXItedit. For each address space in which your device has memory, you must create a separate pseudo-device entry with a logical address between 256 and 511.
For example, a VME device with memory in both A24 and A32 spaces requires two entries. You can also specify which interrupt levels the device uses. VXI and VME devices cannot share interrupt levels. You can then

access the device from NI-VISA just as you would a VXI device, by specifying the address space and the offset from the base at which you have configured it. NI-VISA support for VME devices includes the register access operations (both high-level and low-level) and the block-move operations, as well as the ability to receive interrupts.

Debugging A VISA Program

This section contains information on debugging VISA programs. Because of VISA's nature there are more possibilities to consider when debugging VISA problems than when working with standalone drivers. VISA makes calls into NI-VXI, NI-488, or Operating System serial APIs. Therefore, problems that appear in VISA could be related to the driver VISA is calling, and not VISA itself.

If no VISA VIs appear to be working in LabVIEW (including instrument drivers), the first step to take is the VISA Find Resource VI. This VI will run without any other VISA VIs in the block diagram. If this VI produces strange errors such as nonstandard VISA errors, the problem is most likely that the wrong version of VISA is installed or that VISA is not installed correctly. If VISA Find Resource runs correctly, LabVIEW is working correctly with the VISA driver. The next step is to identify what sequence of VIs is producing the error in the LabVIEW program.

If it is a simple sequence of events that is producing the error, a good next step in debugging is to try the same sequence interactively with the VISAIC utility (see the next section). It is generally a good idea to do initial program development interactively. If the interactive utility works successfully but the same sequence in LabVIEW does not, it is an indication that LabVIEW might have a problem interacting with the VISA driver. If the same sequence exhibits the same problem interactively in VISAIC it is possible that a problem exists with one of the drivers VISA is calling. You can use the interactive utilities for these drivers (VIC for NI-VXI and IBIC for NI-488.2) to try to perform the equivalent operations. If the problems persist on this level, it is an indication that there may be a problem with the lower-level driver or its installation.

Debugging Tool for Windows 95/NT

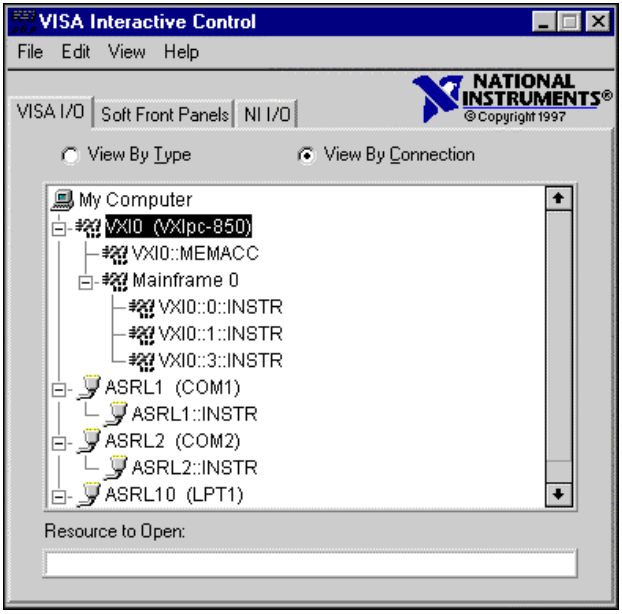
NI Spy tracks the calls your application makes to National Instruments test and measurement (T&M) drivers, including NI-VISA, NI-VXI, and NI-488.2. NI-488.2 users may notice that NI Spy is similar to GPIB Spy.

NI Spy highlights functions that return errors, so you can quickly determine which functions failed during your development. NI Spy can also log your program's calls to these drivers, so you can check them for errors at your convenience.

VISAIC

VISA comes with a utility called VISA Interactive Control (VISAIC) on all platforms that support VISA and LabVIEW, except the Macintosh. This utility provides access to all of VISA's functionality interactively, in an easy-to-use graphical environment. It is a convenient starting point for program development and learning about VISA.

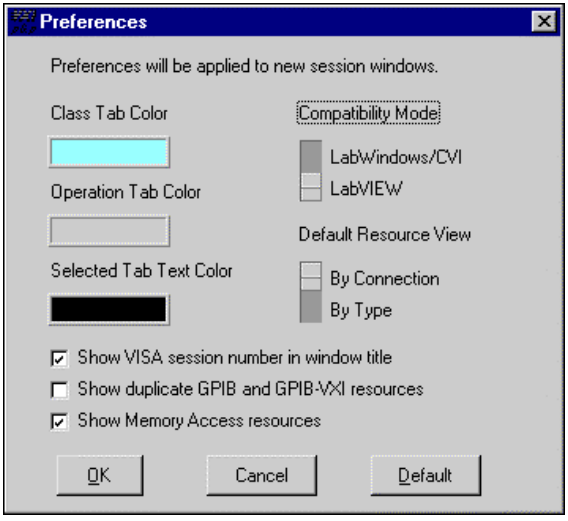
When VISAIC runs it automatically finds all of the available resources in the system and lists the instrument descriptors for each of these resources under the appropriate resource type. The VISAIC opening window is shown in the figure below.



The Soft Front Panels tab of the main VISAIC panel will give you the option to launch the soft front panels of any VXI*plug&play* instrument drivers that have been installed on the system.

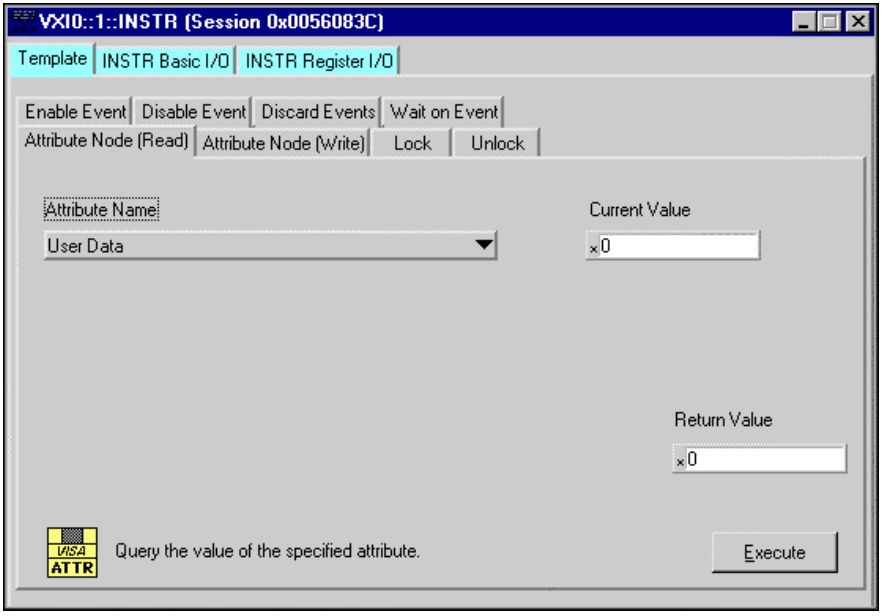
The NI I/O tab will give you the option to launch the NI-VXI interactive utility or the NI-488 interactive utility. This gives you convenient links into the interactive utilities for the drivers VISA calls in case you would like to try debugging at this level.

Double clicking on any of the instrument descriptors shown in the VISAIC window will open a session to that instrument. Opening a session to the instrument produces a window with a series of tabs for interactively running VISA commands. The exact appearance of these tabs depends on which compatibility mode VISAIC is in. To access the compatibility mode and other VISAIC preferences select **Edit»Preferences…** to bring up the window shown below.



The VISA implementations are slightly different in LabVIEW and LabWindows/CVI. These differences are reflected in the operation tabs that are shown when you open a session to a resource. By default the compatibility mode is set to Labwindows/CVI, and you should change this to LabVIEW. Once the preferences are changed the new preferences will take effect for any session that is opened later.

When a session to a resource is opened interactively a window similar to the one shown below will appear.



There are three main tabs that appear in the window. The initial tab is the template tab that contains all of the operations dealing with events, properties, and locks. Notice that there is a different tab for each of these operations under the main tab. The other main tabs are the INSTR Basic I/O and INSTR Register I/O. The Basic I/O tab contains the basic operations for message-based instruments while the Register I/O tab contains the basic operations for register-based instruments. The Register I/O tab only appears for VXI instruments.

9

Introduction to LabVIEW GPIB Functions

This chapter explains how the General Purpose Interface Bus (GPIB) operates and the difference between the IEEE 488 and IEEE 488.2 interfaces.

There are two GPIB standards—IEEE 488 and IEEE 488.2. Hewlett-Packard designed the GPIB (originally called the HP-IB) to interconnect and control its line of programmable instruments. The GPIB was soon applied to other applications such as intercomputer communication and peripheral control because of its 1 Mbytes/s maximum data transfer rates. It was later accepted as IEEE Standard 488-1975 and has since evolved into ANSI/IEEE Standard 488.2-1987. The versatility of the system prompted the name General Purpose Interface Bus.

National Instruments brought the GPIB to users of non-Hewlett-Packard computers and devices, specializing in both high-performance, high-speed hardware interfaces and comprehensive, full-function software. The GPIB functions for LabVIEW follow the IEEE 488.2 specification.

Types of Messages

The GPIB carries device-dependent messages and interface messages.

- Device-dependent messages, often called *data* or *data messages*, contain device-specific information such as programming instructions, measurement results, machine status, and data files.
- Interface messages manage the bus itself. They are usually called *commands* or *command messages*. Interface messages perform such tasks as initializing the bus, addressing and unaddressing devices, and setting device modes for remote or local programming.

Do not confuse the term *command* as used here with some device instructions, which can also be called commands. These device-specific instructions are actually data messages.

The ANSI/IEEE Standard 488.2-1987 expanded on the earlier IEEE 488.1 standard to describe exactly how the Controller should manage the GPIB, including the standard messages that compliant devices should understand, the mechanisms for reporting device errors and other status information, and the various protocols that discover and configure compliant devices connected to the bus.

IEEE 488.2 has the ability to monitor any of the bus lines at any time is crucial for detecting active devices (Talkers and Listeners) on the GPIB. GPIB devices can be Talkers, Listeners, and/or Controllers. A digital voltmeter, for example, is a Talker and may be a Listener as well. A Talker sends data messages to one or more Listeners. The Controller manages the flow of information on the GPIB by sending commands to all devices.

The GPIB is like an ordinary computer bus, except that the computer has its circuit cards interconnected via a backplane bus, whereas the GPIB has stand-alone devices interconnected via a cable bus.

The role of the GPIB Controller is similar to the role of the CPU of a computer, but a better analogy is to the switching center of a city telephone system. The switching center (Controller) monitors the communications network (GPIB). When the center (Controller) notices that a party (device) wants to make a call (send a data message), it connects the caller (Talker) to the receiver (Listener).

The Controller addresses a Talker and a Listener before the Talker can send its message to the Listener. After the Talker transmits the message, the Controller may unaddress both devices.

Some bus configurations do not require a Controller. For example, one device may always be a Talker (called a Talk-only device) and there may be one or more Listen-only devices.

A Controller is necessary when you must change the active or addressed Talker or Listener. A computer usually handles the Controller function.

With the GPIB board and its software, your personal computer plays all three roles:

- Controller—to manage the GPIB
- Talker—to send data
- Listener—to receive data

The Controller-In-Charge and System Controller

Although there can be multiple Controllers on the GPIB, only one Controller at a time is active or Controller-In-Charge (CIC). You can pass active control from the current CIC to an idle Controller. Only one device on the bus—the System Controller—can make itself the CIC. The GPIB board is usually the System Controller.

Compatible GPIB Hardware

The following National Instruments GPIB hardware products are compatible with LabVIEW:

LabVIEW for Windows 95 and Windows 95-Japanese

- AT-GPIB/TNT, AT-GPIB/TNT (PnP), AT-GPIB/TNT+, PCI-GPIB
- PCMCIA-GPIB, PCMCIA-GPIB+
- GPIB-ENET
- EISA-GPIB
- VXIpc Model 850
- NEC-GPIB/TNT, NEC-GPIB/TNT (PnP)
- GPIB-PCII/IIA
- PC/104-GPIB
- CPCI-GPIB
- GPIB-ENET
- PMC-GPIB

LabVIEW for Windows NT

- AT-GPIB, AT-GPIB/TNT
- PCMCIA-GPIB
- PCI-GPIB
- VXIpc Model 850
- GPIB-ENET

LabVIEW for Windows 3.1

- AT-GPIB, AT-GPIB/TNT, AT-GPIB/TNT (PnP), AT-GPIB/TNT+, PCI-GPIB
- PCMCIA-GPIB, PCMCIA-GPIB+
- GPIB-ENET
- EISA-GPIB
- VXIpc Model 850
- NEC-GPIB/TNT (Japanese), NEC-GPIB/TNT (PnP) (Japanese), GPIB-PCII/IIA
- GPIB-232CT-A
- GPIB-485CT-A
- GPIB-1284CT
- PCII/IIA
- STD-GPIB
- EXM-GPIB
- MC-GPIB

LabVIEW for Mac OS

- PCI-GPIB
- NB-GPIB/TNT, NB-GPIB-P/TNT
- PCMCIA-GPIB
- LC-GPIB
- GPIB-ENET
- GPIB-232CT-A
- GPIB-SCSI-A
- PC/104-GPIB
- NB-DMA2800 (Traditional GPIB functions only)

LabVIEW for HP-UX

- GPIB-ENET
- EISA-GPIB
- AT-GPIB/TNT

LabVIEW for Sun

- GPIB-ENET
- GPIB-SCSI-A
- SB-GPIB/TNT

LabVIEW for Concurrent PowerMAX

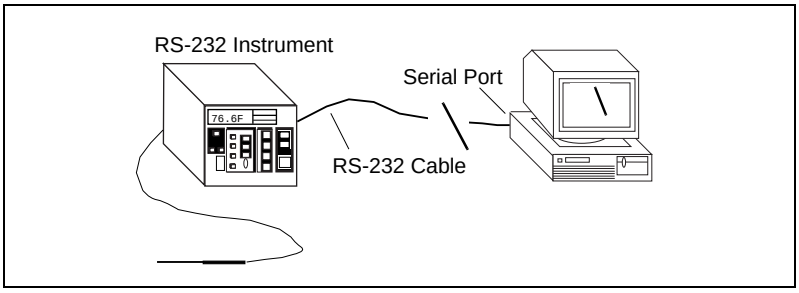
- GPIB-1014
- GPIB-1014D
- GPIB-1014P
- GPIB-1014DP

10

Serial Port VIs

This chapter describes the VIs for serial port operations and explains the important factors that affect serial communication.

Serial communication is a popular means of transmitting data between a computer and a peripheral device such as a programmable instrument or even another computer. 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 you must transfer data over long distances.



Serial communication is popular because most computers have one or two serial ports. Many GPIB instruments also are available with serial ports. A limitation of serial communication, however, is that a serial port can communicate with only one device.

Some peripheral devices require characters to terminate strings of data sent to them. Common terminating characters are a carriage return, a line feed, or a semicolon. Consult the device manual to determine if a terminating character is needed.

For examples of how to use the Serial Port VIs, see `examples\instr\smplserl.llb`.

Handshaking Modes

A common problem in serial communications is ensuring that both sender and receiver keep up with data transmission. The serial port driver can buffer incoming/outgoing information, but that buffer is of a finite size. When it becomes full, the computer ignores new data until you have read enough data out of the buffer to make room for new information.

Handshaking helps prevent this buffer from overflowing. With handshaking, the sender and the receiver notify each other when their buffers fill up. The sender can then stop sending new information until the other end of the serial communication is ready for new data.

You can perform two kinds of handshaking in LabVIEW—software handshaking and hardware handshaking. You can turn both of these forms of handshaking on or off using the Serial Port Init VI. By default, the VIs do not use handshaking.

Software Handshaking—XON/XOFF

XON/XOFF is a software handshaking protocol you can use to avoid overflowing serial port buffers. When the receive buffer is nearly full, the receiver sends XOFF (<Ctrl-S> [decimal 19]) to tell the other device to stop sending data. When the receive buffer is sufficiently empty, the receiver sends XON (<Ctrl-Q> [decimal 17]) to indicate that transmission can begin again. When you enable XON/XOFF, the devices always interpret <Ctrl-Q> and <Ctrl-S> as XON and XOFF characters, never as data. When you disable XON/XOFF, you can send <Ctrl-Q> and <Ctrl-S> as data. Do not use XON/XOFF with binary data transfers because <Ctrl-Q> or <Ctrl-S> may be embedded in the data, and the devices will interpret them as XON and XOFF instead of as data.

Error Codes

You can connect the **error code** parameter to one of the error handler VIs. These VIs can describe the error and give you options on how to proceed when an error occurs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Port Number

Windows 95/NT and 3.*x*

When you use the serial port VIs under Windows 95/NT and Windows 3.*x*, the **port number** parameter can have the following values:

0:	COM1	5:	COM6	10:	LPT1
1:	COM2	6:	COM7	11:	LPT2
2:	COM3	7:	COM8	12:	LPT3
3:	COM4	8:	COM9	13:	LPT4
4:	COM5				

When you use the serial port VIs under Windows 95 or Windows NT, the **port number** parameter is 0 for COM1, 1 for COM2, and so on.

Macintosh

On the Macintosh, port 0 is the modem, using the drivers `.ain` and `.aout`. Port 1 is the printer, using the drivers `.bin` and `.bout`. To get more ports on a Macintosh, you must install other boards, with the accompanying drivers.

UNIX

On a Sun SPARCstation under Solaris 1 and on Concurrent PowerMAX, the **port number** parameter for the serial port VIs is 0 for `/dev/ttya`, 1 for `/dev/ttyb`, and so on. Under Solaris 2, port 0 refers to `/dev/cua/a`, 1 to `/dev/cua/b`, and so on. Under HP-UX port number 0 refers to `/dev/tty00`, 1 to `/dev/tty01`, and so on.

On Concurrent PowerMAX, port 0 refers to `/dev/console`, Port 1 refers to `/dev/tty1`, Port 2 refers to `/dev/tty2`, and so on.

Because other vendor's serial port boards can have arbitrary device names, LabVIEW has developed an easy interface to keep the numbering of ports simple. In LabVIEW for Sun, HP-UX, and Concurrent PowerMAX, a configuration option exists to tell LabVIEW how to address the serial ports. LabVIEW supports any board that uses standard UNIX devices. Some manufacturers suggest using `cua` rather than `tty` device nodes with their boards. LabVIEW can address both types of nodes.

The file `.labviewrc` contains the LabVIEW configuration options. To set the devices the serial port VIs use, set the configuration option `labview.serialDevices` to the list of devices you intend to use.

For example, the default is:

```
labview.serialDevices:/dev/ttya:/dev/ttyb:/dev/ttyc:...
:/dev/ttyz.
```

Note ***This requires that any third party serial board installation include a method of creating a standard*** `/dev` ***file (node) and that the user knows the name of that file.***

Part III

Analysis

This section contains basic information on analysis of post acquisition data, signal processing, signal generation, linear algebra, curve fitting, probability, and statistics.

Part III, *Analysis*, contains the following chapters.

- Chapter 11, *Introduction to Analysis in LabVIEW*, introduces concepts that apply to all analysis applications, including supported functionality, notation and naming conventions, and sampling signal methods.
- Chapter 12, *Signal Generation*, explains how to produce signals using normalized frequency and how to build a simulated function generator.
- Chapter 13, *Digital Signal Processing*, describes the fundamentals of the Fast Fourier Transform (FFT) and the Discrete Fourier Transform (DFT) and how they are used in spectral analysis.
- Chapter 14, *Smoothing Windows*, explains how using windows prevents spectral leakage and improves the analysis of acquired signals.
- Chapter 15, *Spectrum Analysis and Measurement*, shows how to determine the amplitude and phase spectrum, develop a spectrum analyzer, and determine the total harmonic distortion (THD) of your signals.
- Chapter 16, *Filtering*, explains how to filter unwanted frequencies from signals using infinite impulse response filters (IIR), finite impulse response filters (FIR), and nonlinear filters.
- Chapter 17, *Curve Fitting*, shows how to extract information from a data set to obtain a functional description.

- Chapter 18, *Linear Algebra*, explains how to perform matrix computation and analysis.
- Chapter 19, *Probability and Statistics*, explains some fundamental concepts on probability and statistics, and shows how to use these concepts in solving real-world problems.

11

Introduction to Analysis in LabVIEW

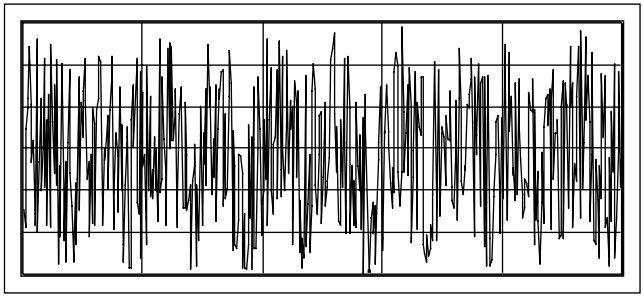
Digital signals are everywhere in the world around us. Telephone companies use digital signals to represent the human voice. Radio, TV, and hi-fi sound systems are all gradually converting to the digital domain because of its superior fidelity, noise reduction, and signal processing flexibility. Data is transmitted from satellites to earth ground stations in digital form. NASA's pictures of distant planets and outer space are often processed digitally to remove noise and extract useful information. Economic data, census results, and stock market prices are all available in digital form. Because of the many advantages of digital signal processing, analog signals are also converted to digital form before they are processed with a computer.

This chapter provides a background in basic digital signal processing and an introduction to the LabVIEW Analysis Library, which consists of hundreds of VIs for signal processing and analysis.

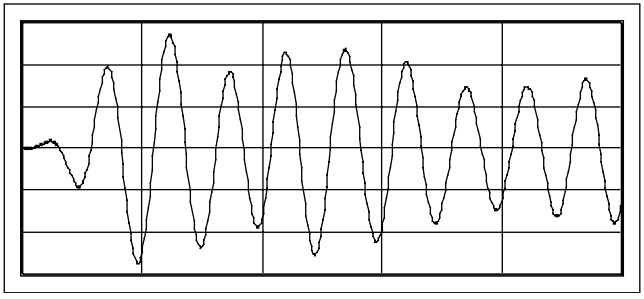
The Importance of Data Analysis

Modern, high-speed floating-point numerical and digital signal processors have become increasingly important to real-time and analysis systems. A few of the many possible applications include biomedical data processing, speech synthesis and recognition, and digital audio and image processing.

The importance of integrating analysis libraries into engineering stations is that the *raw* data, as shown in the following figure, does not always immediately convey useful information. Often you must transform the signal, remove noise disturbances, correct for data corrupted by faulty equipment, or compensate for environmental effects, such as temperature and humidity.



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 following figure shows the processed data.



The LabVIEW block diagram programming approach and the extensive set of LabVIEW analysis VIs simplify the development of analysis applications.

The LabVIEW analysis VIs give you the most recent data analysis techniques using VIs that you can wire together. Instead of worrying about implementation details for analysis routines, as you do in conventional programming languages, you can concentrate on solving your data analysis problems.

Full Development System

The base analysis VI library is a subset of the advanced analysis VI library. The base analysis library includes VIs for statistical analysis, linear algebra, and numerical analysis. The advanced analysis library includes more VIs in these areas as well as VIs for signal generation, time and frequency-domain algorithms, windowing routines, digital filters, evaluations, and regressions.

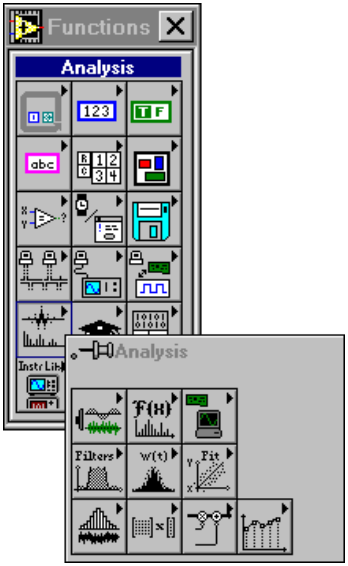
If the VIs in the base analysis library do not satisfy your needs, then you can add the LabVIEW Advanced Analysis Libraries to the LabVIEW Base Package. Once you upgrade, you will have all the analysis tools available in the Full Development System.

Analysis VI Overview

Once the analog signal has been converted to digital form by the A/D Converter (ADC) and is available in your computer as a digital signal (a set of samples), you will usually want to process these samples in some way. The processing could be to determine the characteristics of the system from which the samples were obtained, to measure certain features of the signal, or to convert them into a form suitable for human understanding.

The Analysis library contains VIs to perform extensive numerical analysis, signal generation and signal processing, curve fitting, measurement, and other analysis functions. The Analysis Library, included in the LabVIEW full development system, is a key component in building a virtual instrumentation system. Besides containing the analysis functionality found in many math packages, it also features many unique signal processing and measurement functions that are designed exclusively for the instrumentation industry.

The LabVIEW Analysis VIs are available in the **Analysis** subpalette of the **Functions** Palette in LabVIEW or BridgeVIEW.



There are 10 analysis VI libraries. The main categories are:

Signal Generation: VIs that generate digital patterns and waveforms.

Digital Signal Processing: VIs that perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms such as the Hartley and Hilbert transforms.

Measurement Functions: VIs that perform measurement-oriented functions such as single-sided spectrums, scaled windowing, and peak power and frequency estimation.

Digital Filters: VIs that perform IIR, FIR, and nonlinear digital filtering functions.

Windowing Functions: VIs that perform data windowing.

Probability and Statistics Functions: VIs that perform descriptive statistics functions, such as identifying the mean or the standard deviation of a set of data, as well as inferential statistics functions for probability and analysis of variance (ANOVA).

Curve Fitting Functions: VIs that perform curve fitting functions and interpolations.

Linear Algebra Functions: VIs that perform algebraic functions for real and complex vectors and matrices.

Array Operations: VIs that perform common, one- and two-dimensional numerical array operations, such as linear evaluation and scaling.

Additional Numerical Methods: VIs that use numerical methods to perform root-finding, numerical integration, and peak detection.

In these chapters, you will learn how to use the VIs from the analysis library to build a function generator and a simple, yet practical, spectrum analyzer. You will also learn how to use digital filters, the purpose of windowing and the advantages of different types of windows, how to perform simple curve-fitting tasks, and much more. The activities in these chapters require the LabVIEW/BridgeVIEW full development system. For the more adventurous, an extensive set of examples that demonstrate how to use the analysis VIs can be found in the `labview\examples\analysis` folder.

In addition to the Analysis library, National Instruments also offers many analysis add-ons that make LabVIEW one of the most powerful analysis software packages available. These add-ons include the *Joint Time-Frequency Analysis Toolkit*, which includes the National Instruments award-winning Gabor spectrogram algorithm that analyzes time-frequency features not easily obtained by conventional Fourier analysis; the *G Math Toolkit* that offers extended math functionality like a formula parser, routines for optimization and solving differential equations, numerous types of 2D and 3D plots, and more; the *Digital Filter Design Toolkit*; and many others.

Notation and Naming Conventions

To help you identify the type of parameters and operations, this section of the manual uses the following notation and naming conventions unless otherwise specified in a VI description. Although there are a few scalar functions and operations, most of the analysis VIs process large blocks of data in the form of one-dimensional arrays (or vectors) and two-dimensional arrays (or matrices).

Normal lower case letters represent scalars or constants. For example,

$$a,$$
$$\pi,$$
$$b = 1.234.$$

Capital letters represent arrays. For example,

$$X,$$
$$A,$$
$$Y = a\,X + b.$$

In general, *X* and *Y* denote 1D arrays, and *A*, *B*, and *C* represent matrices.

Array indexes in LabVIEW are zero-based. The index of the first element in the array, regardless of its dimension, is zero. The following sequence of numbers represents a 1D array *X* containing *n* elements.

$$X = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$$

The following scalar quantity represents the i^{th} element of the sequence *X*.

$$x_i, \quad 0 \le i < n$$

The first element in the sequence is x_0 and the last element in the sequence is x_{n-1}, for a total of *n* elements.

The following sequence of numbers represents a 2D array containing *n* rows and *m* columns.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0m-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1m-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-10} & a_{n-11} & a_{n-2} & \cdots & a_{n-1m-1} \end{bmatrix}$$

The total number of elements in the 2D array is the product of *n* and *m*. The first index corresponds to the row number, and the second index corresponds to the column number. The following scalar quantity represents the element located on the i^{th} row and the j^{th} column.

$$a_{ij},\ 0 \le i < n \text{ and } 0 \le j < m$$

The first element in *A* is $a_{0\,0}$ and the last element is $a_{n-1\,m-1}$.

Unless otherwise specified, this manual uses the following simplified array operation notations.

Setting the elements of an array to a scalar constant is represented by

$$X = a,$$

which corresponds to the sequence

$$X = \{a, a, a, \ldots, a\}$$

and is used instead of

$$x_i = a,$$

for

$$i = 0, 1, 2, \ldots, n-1.$$

Multiplying the elements of an array by a scalar constant is represented by

$$Y = a\,X,$$

which corresponds to the sequence

$$Y = \{a\,x_0, a\,x_1, a\,x_2, \ldots, a\,x_{n-1}\}$$

and is used instead of

$$y_i = a\,x_i,$$

for

$$i = 0, 1, 2, \ldots, n-1.$$

Similarly, multiplying a 2D array by a scalar constant is represented by

$$B = k\,A,$$

which corresponds to the sequence

$$B = \begin{bmatrix} ka_{00} & ka_{01} & ka_{02} & \ldots & ka_{0m-1} \\ ka_{10} & ka_{11} & ka_{12} & \ldots & ka_{1m-1} \\ ka_{20} & ka_{21} & ka_{22} & \ldots & ka_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ ka_{n-10} & ka_{n-11} & ka_{n-12} & \ldots & ka_{n-1m-1} \end{bmatrix}$$

and is used instead of

$$b_{ij} = k\,a_{ij},$$

for

$$i = 0, 1, 2, \ldots, n-1$$

and

$$j = 0, 1, 2, \ldots, m-1.$$

An array with no elements is an empty array and is represented by

$$\text{Empty} = \text{NULL} = \emptyset = \{\ \}\ .$$

In general, operations on empty arrays result in empty output arrays or undefined results.

Data Sampling

Sampling Signals

To use digital signal processing techniques, you must first convert an analog signal into its digital representation. In practice, this is implemented by using an analog-to-digital (A/D) converter. Consider an analog signal $x(t)$ that is sampled every Δt seconds. The time interval Δt is known as the *sampling interval* or *sampling period*. Its reciprocal, $1/\Delta t$, is known as the *sampling frequency*, with units of samples/second. Each of the discrete values of x(t) at t = 0, Δt, $2\Delta t$, $3\Delta t$, etc., is known as a *sample*. Thus, x(0), x(Δt), x($2\Delta t$), ..., are all samples. The signal $x(t)$ can thus be represented by the discrete set of samples

$$\{x(0), x(\Delta t), x(2\Delta t), x(3\Delta t), \ldots, x(k\Delta t), \ldots\}.$$

Figure 11-1 below shows an analog signal and its corresponding sampled version. The sampling interval is Δt. Observe that the samples are defined at discrete points in time.

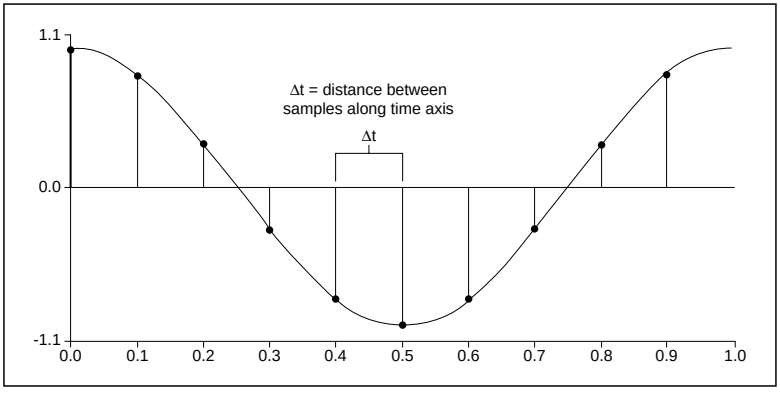


Figure 11-1. Analog Signal and Corresponding Sampled Version

The following notation represents the individual samples:

$$x[i] = x(i\Delta t),$$

for

$$i = 0, 1, 2, \ldots$$

If N samples are obtained from the signal $x(t)$, then $x(t)$ can be represented by the sequence

$$X = \{x[0], x[1], x[2], x[3], \ldots, x[N-1]\}$$

This is known as the *digital representation* or the *sampled version* of $x(t)$. Note that the sequence $X = \{x[i]\}$ is indexed on the integer variable i, and does not contain any information about the sampling rate. So by knowing just the values of the samples contained in X, you will have no idea of what the sample rate is.

Sampling Considerations

A/D converters (ADCs) are an integral part of National Instruments DAQ boards. One of the most important parameters of an analog input system is the rate at which the DAQ board samples an incoming signal. The sampling rate determines how often an analog-to-digital (A/D) conversion takes place. A fast sampling rate acquires more points in a given time and can therefore often form a better representation of the original signal than a slow sampling rate. Sampling too slowly may result in a poor representation of your analog signal. Figure 11-2 shows an adequately sampled signal, as well as the effects of undersampling. The effect of undersampling is that the signal appears as if it has a different frequency than it truly does. This misrepresentation of a signal is called an *alias*.

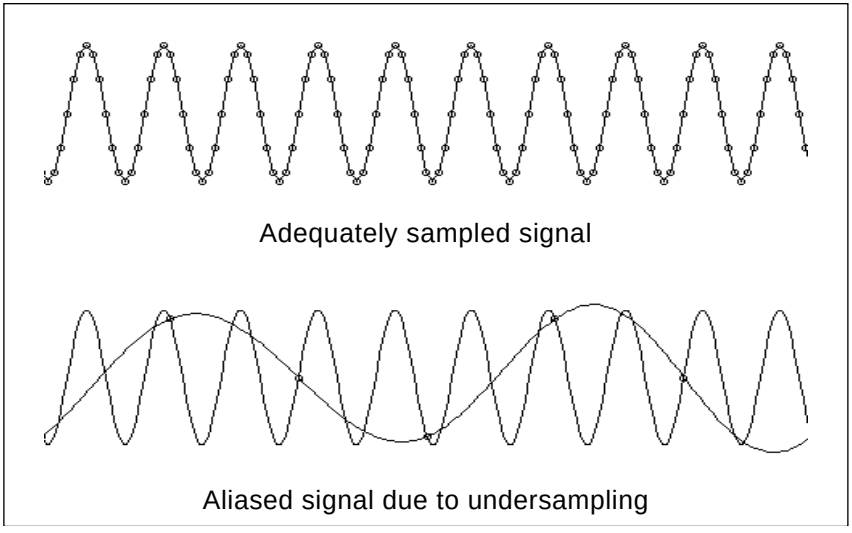


Figure 11-2. Aliasing Effects of an Improper Sampling Rate

According to the *Nyquist theorem*, to avoid aliasing you must sample at a rate greater than twice the maximum frequency component in the signal you are acquiring. For a given sampling rate, the maximum frequency that can be represented accurately, without aliasing, is known as the Nyquist frequency. The Nyquist frequency is one half the sampling frequency. Signals with frequency components above the Nyquist frequency will appear aliased between DC and the Nyquist frequency. The alias frequency is the absolute value of the difference between the frequency of the input signal and the closest integer multiple of the sampling rate. Figures 11-3 and 11-4 illustrate this phenomenon. For example, assume *f*s, the sampling frequency, is 100 Hz. Also, assume the input signal contains the following frequencies—25 Hz, 70 Hz, 160 Hz, and 510 Hz. These frequencies are shown in Figure 11-3.

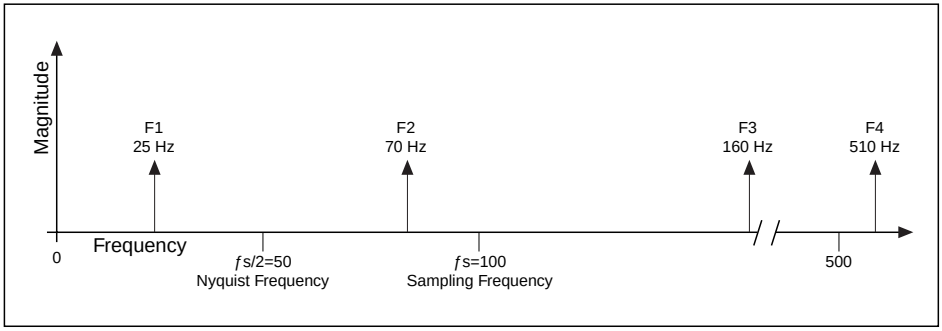


Figure 11-3. Actual Signal Frequency Components

In Figure 11-4, we see that frequencies below the Nyquist frequency (*f*s/2=50 Hz) are sampled correctly. Frequencies above the Nyquist frequency appear as aliases. For example, F1 (25 Hz) appears at the correct frequency, but F2 (70 Hz), F3 (160 Hz), and F4 (510 Hz) have aliases at 30 Hz, 40 Hz, and 10 Hz, respectively. To calculate the alias frequency, use the following equation:

Alias Freq. = ABS (Closest Integer Multiple of Sampling Freq. – Input Freq.)

where ABS means "the absolute value." For example,

Alias F2 = |100 – 70| = 30 Hz
Alias F3 = |(2)100 – 160| = 40 Hz
Alias F4 = |(5)100 – 510| = 10 Hz

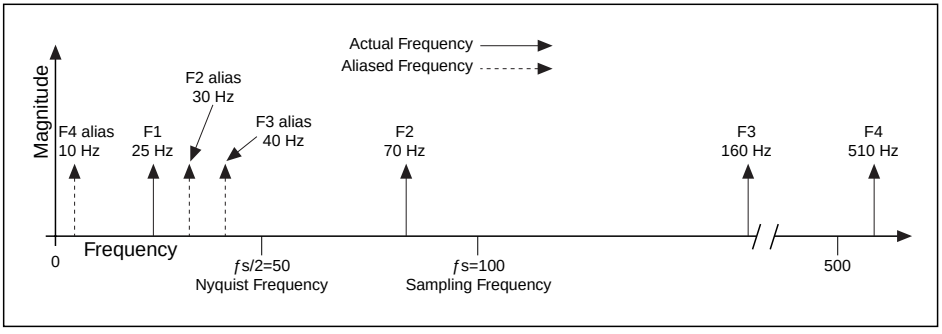


Figure 11-4. Signal Frequency Components and Aliases

A question often asked is, "How fast should I sample?" Your first thought may be to sample at the maximum rate available on your DAQ board. However, if you sample very fast over long periods of time, you may not have enough memory or hard disk space to hold the data. Figure 11-5 shows the effects of various sampling rates. In case a, the sine wave of frequency f is sampled at the same frequency fs (samples/sec) = f (cycles/sec), or at 1 sample per cycle. The reconstructed waveform appears as an alias at DC. As you increase the sampling to 7 samples/4 cycles, as in case b, the waveform increases in frequency, but aliases to a frequency less than the original signal (3 cycles instead of 4). The sampling rate in case b is fs = 7/4 f. If you increase the sampling rate to fs = 2f, the digitized waveform has the correct frequency (same number of cycles), and can be reconstructed as the original sinusoidal wave, as shown in case c. For time-domain processing, it may be important to increase your sampling rate so that the samples more closely represent the original signal. By increasing

the sampling rate to well above f, say to fs=10f, or 10 samples/cycle, you can accurately reproduce the waveform, as shown in case d.

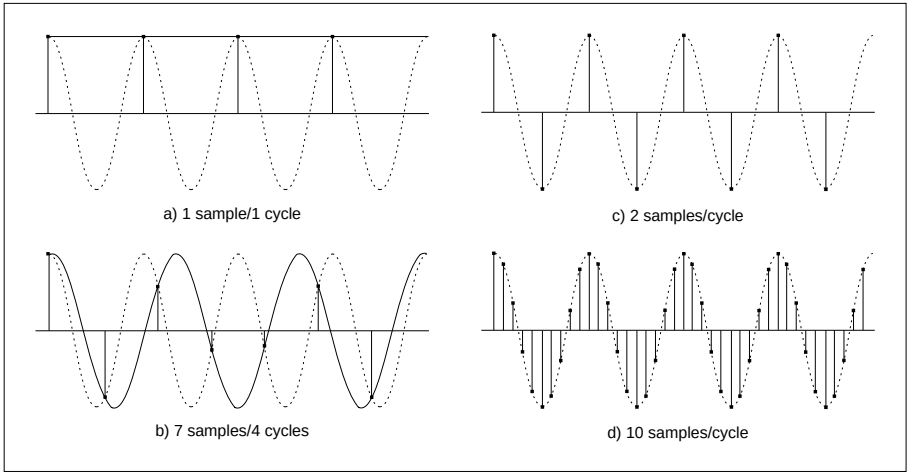


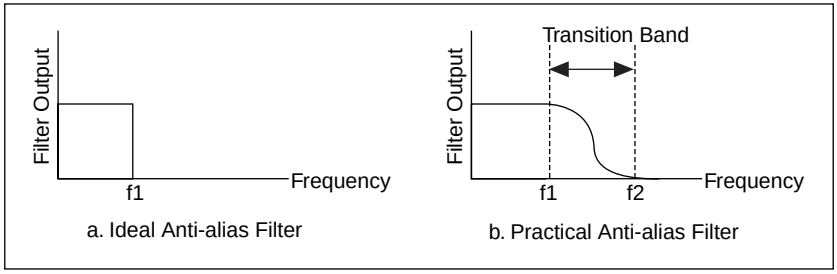
Figure 11-5. Effects of Sampling at Different Rates

Why Do You Need Anti-Aliasing Filters?

We have seen that the sampling rate should be at least twice the maximum frequency of the signal that we are sampling. In other words, the maximum frequency of the input signal should be less than or equal to half of the sampling rate. But how do you ensure that this is definitely the case in practice? Even if you are sure that the signal being measured has an upper limit on its frequency, pickup from stray signals (such as the powerline frequency or from local radio stations) could contain frequencies higher than the Nyquist frequency. These frequencies may then alias into the desired frequency range and thus give us erroneous results.

To be completely sure that the frequency content of the input signal is limited, a low pass filter (a filter that passes low frequencies but attenuates the high frequencies) is added before the sampler and the ADC. This filter is called an *anti-alias* filter because by attenuating the higher frequencies (greater than Nyquist), it prevents the aliasing components from being sampled. Because at this stage (before the sampler and the ADC) we are still in the analog world, the anti-aliasing filter is an analog filter.

An ideal anti-alias filter is as shown in figure (a) below.



It passes all the desired input frequencies (below *f1*) and cuts off all the undesired frequencies (above *f1*). However, such a filter is not physically realizable. In practice, filters look as shown in figure (b) above. They pass all frequencies < *f1*, and cut-off all frequencies > *f2*. The region between *f1* and *f2* is known as the *transition band*, which contains a gradual attenuation of the input frequencies. Although you want to pass only signals with frequencies < *f1*, those signals in the transition band could still cause aliasing. Therefore, in practice, the sampling frequency should be greater than two times the highest frequency in the transition band. So, this turns out to be more than two times the maximum input frequency (*f1*). That is one reason why you may see that the sampling rate is more than twice the maximum input frequency. We will see in a later section how the transition band of the filter depends on the filter type being designed.

Why Use Decibels?

On some instruments, you will see the option of displaying the amplitude in a linear or decibel (dB) scale. The linear scale shows the amplitudes as they are, whereas the decibel scale is a transformation of the linear scale into a logarithmic scale. We will now see why this transformation is necessary.

Suppose that you want to display a signal with very large as well as very small amplitudes. Let us assume you have a display of height 10 cm, and will utilize the entire height of the display for the largest amplitude. So, if the largest amplitude in the signal is 100 V, a height of 1 cm of the display corresponds to 10 V. If the smallest amplitude of the signal is 0.1 V, this corresponds to a height of only 0.1 mm. This will barely be visible on the display.

To see all the amplitudes, from the largest to the smallest, you need to change the amplitude scale. Alexander Graham Bell invented a unit, the Bell, which is logarithmic, compressing large amplitudes and expanding

the small amplitudes. However, the Bell was too big of a unit, so commonly the decibel (1/10th of a Bell) is used. The decibel (dB) is defined as

$$\text{one dB} = 10 \log_{10} (\text{Power Ratio}) = 20 \log_{10} (\text{Voltage Ratio})$$

The following table shows the relationship between the decibel and the Power and Voltage Ratios.

dB	Power Ratio	Voltage Ratio
+40	10000	100
+20	100	10
+6	4	2
+3	2	1.4
0	1	1
–3	1/2	1/1.4
–6	1/4	1/2
–20	1/100	1/10
–40	1/10000	1/100

Thus, you see that the dB scale is useful in compressing a wide range of amplitudes into a small set of numbers. The decibel scale is often used in sound and vibration measurements and in displaying frequency domain information.

12

Signal Generation

This chapter explains how to produce signals using normalized frequency and how to build a simulated function generator. For examples of how to use the signal generation VIs, see the examples located in `examples\analysis\sigxmpl.llb`.

You will learn how to use the VIs in the analysis library to generate many different types of signals. Some of the applications for signal generation are:

- Simulating signals to test your algorithm when real-world signals are not available (for example, when you do not have a DAQ board for obtaining real-world signals, or when access to real-world signals is not possible).
- Generating signals to apply to a D/A converter.

Normalized Frequency

In the analog world, a signal frequency is measured in Hz or cycles per second. But the digital system often uses a digital frequency, which is the ratio between the analog frequency and the sampling frequency:

digital frequency = analog frequency / sampling frequency

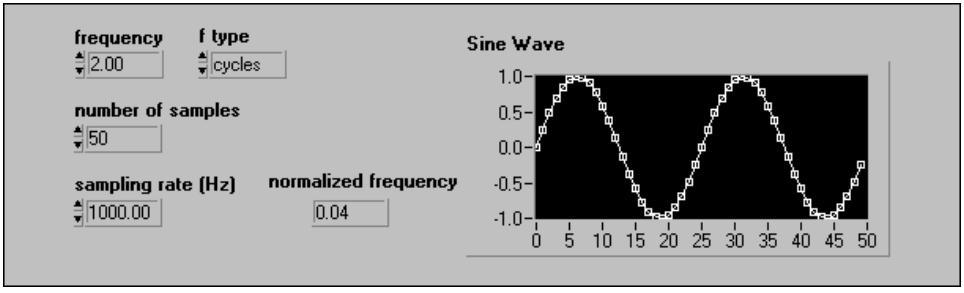
This digital frequency is known as the *normalized* frequency. Its units are cycles/sample.

Some of the Signal Generation VIs use an input frequency control, *f*, that is assumed to use *normalized frequency* units of *cycles per sample*. This frequency ranges from 0.0 to 1.0, which corresponds to a real frequency range of 0 to the sampling frequency *fs*. This frequency also wraps around 1.0, so that a normalized frequency of 1.1 is equivalent to 0.1. As an example, a signal that is sampled at the Nyquist rate (*fs/2*) means that it is sampled twice per cycle (that is, two samples/cycle). This will correspond to a normalized frequency of 1/2 cycles/sample = 0.5 cycles/sample. The reciprocal of the normalized frequency, *1/f*, gives you the number of times that the signal is sampled in one cycle.

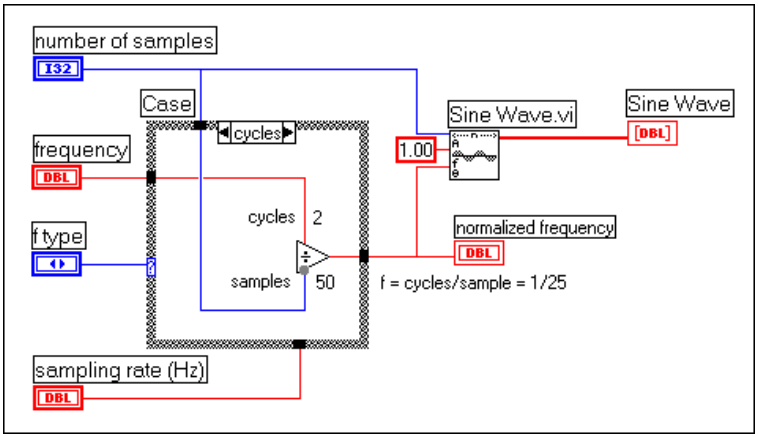
When you use a VI that requires the normalized frequency as an input, you must convert your frequency units to the normalized units of cycles/sample. You must use these normalized units with the following VIs.

- Sine Wave
- Square Wave
- Sawtooth Wave
- Triangle Wave
- Arbitrary Wave
- Chirp Pattern

If you are used to working in frequency units of cycles, you can convert cycles to cycles/sample by dividing cycles by the number of samples generated. The following illustration shows the **Sine Wave** VI, which is being used to generate two cycles of a sine wave.



The following illustration shows the block diagram for converting cycles to cycles/sample.

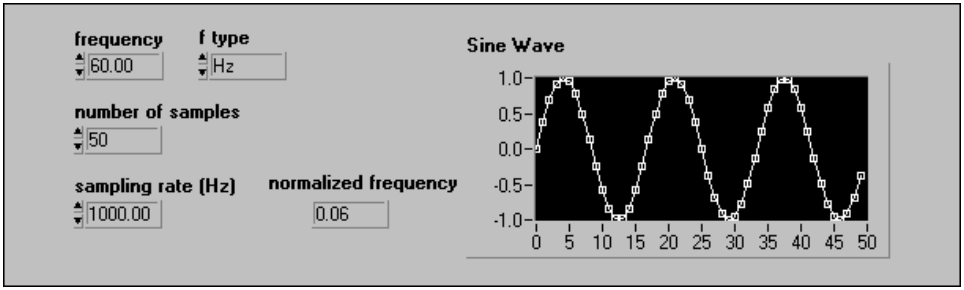


You need only divide the frequency (in cycles) by the number of samples. In the above example, the frequency of 2 cycles is divided by 50 samples, resulting in a normalized frequency of $f = 1/25$ cycles/sample. This means that it takes 25 (the reciprocal of f) samples to generate one cycle of the sine wave.

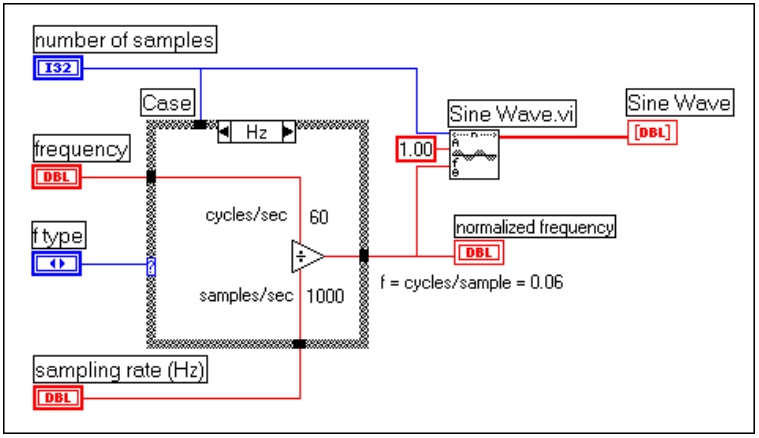
However, you may need to use frequency units of Hz (cycles/s). If you need to convert from Hz (or cycles/s) to cycles/sample, divide your frequency in cycles/s by the sampling rate given in samples/s.

$$\frac{\text{cycles/s}}{\text{samples/s}} = \frac{\text{cycles}}{\text{sample}}$$

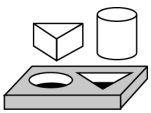
The following illustration shows the **Sine Wave** VI used to generate a 60 Hz sine signal.



The following illustration shows a block diagram for generating a Hertz sine signal. You divide the frequency of 60 Hz by the sampling rate of 1000 Hz to get the *normalized* frequency of f = 0.06 cycles/sample. Therefore, it takes almost 17 (1/0.06) samples to generate one cycle of the sine wave.



The signal generation VIs create many common signals required for network analysis and simulation. You can also use the signal generation VIs in conjunction with National Instruments hardware to generate analog output signals.

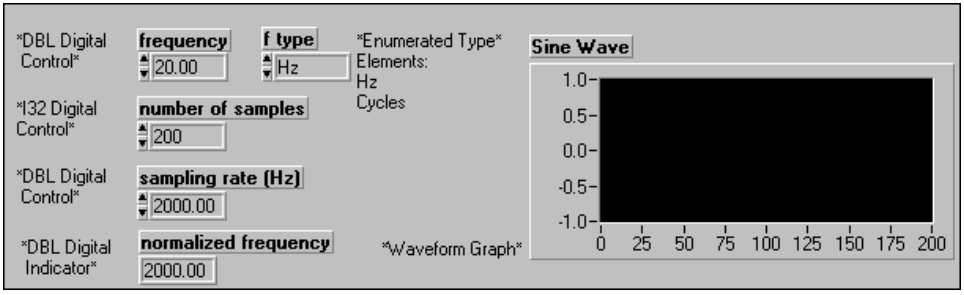


Activity 12-1. Learn More about Normalized Frequency

Your objective is to learn more about normalized frequency by adjusting the frequency, sampling rate, and number of samples and observing the effects on a sine wave.

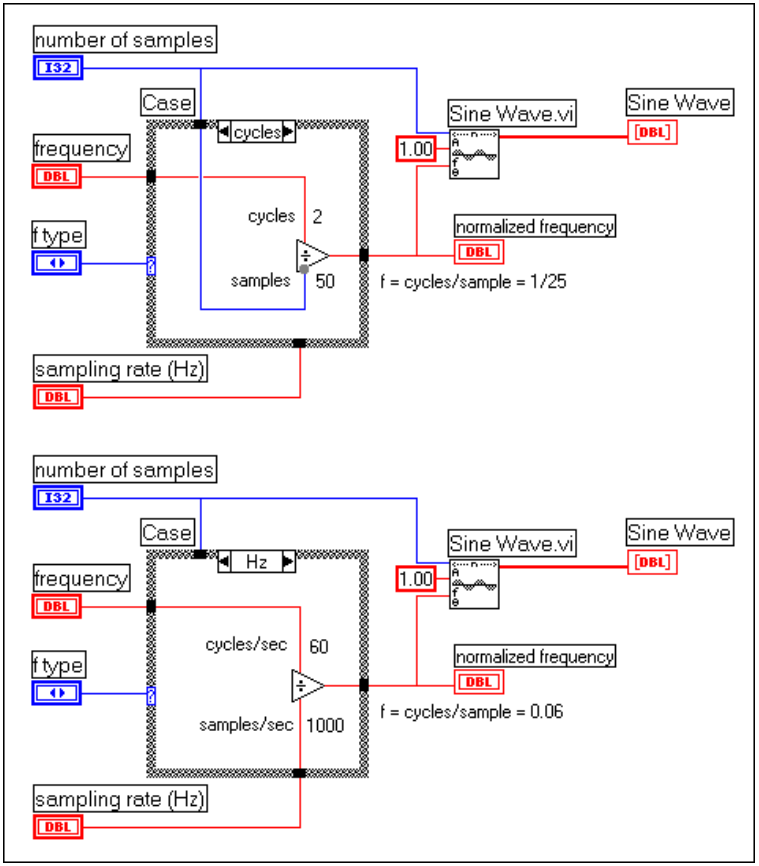
Front Panel

1. Open a new front panel and create the objects as shown in the following illustration.



Block Diagram

2. Build the block diagram shown in the following illustration.

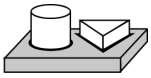


Sine Wave VI (**Analysis»Signal Generation** palette).

3. Save the VI as `Normalized Frequency.vi` in the `LabVIEW\Activity` directory.
4. Select a frequency of 2 cycles (**frequency** = 2 and **ftype** = cycles) and **number of samples** = 100. Run the VI. Note that the plot will show 2 cycles.
5. Increase the **number of samples** to 150, 200, and 250. How many cycles do you see?
6. Now keep the **number of samples** = 100. Increase the number of cycles to 3, 4, and 5. How many cycles do you see?

Thus, when you choose the frequency in terms of cycles, you will see that many cycles of the input waveform on the plot. Note that the sampling rate is irrelevant in this case.

7. Change **ftype** to Hz and **sampling rate (Hz)** to 1,000.
8. Keeping the **number of samples** fixed at 100, change the **frequency** to 10, 20, 30, and 40. How many cycles of the waveform do you see on the plot for each case? Explain your observations.
9. Repeat the above step by keeping the **frequency** fixed at 10 and change the **number of samples** to 100, 200, 300, and 400. How many cycles of the waveform do you see on the plot for each case? Explain your observations.
10. Keep the **frequency** fixed at 20 and the **number of samples** fixed at 200. Change the **sampling rate (Hz)** to 500, 1,000, and 2,000. Make sure you understand the results.



End of Activity 12-1.

Wave and Pattern VIs

You will notice that the names of most of the signal generation VIs have the word *wave* or *pattern* in them. There is a basic difference in the operation of the two different types of VIs. It has to do with whether or not the VI can keep track of the phase of the signal that it generates each time it is called.

Phase Control

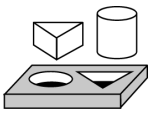
The *wave* VIs have a *phase in* control where you can specify the initial phase (in degrees) of the first sample of the generated waveform. They also have a *phase out* indicator that specifies what the phase of the next sample of the generated waveform is going to be. In addition, a *reset phase* control decides whether or not the phase of the first sample generated when the *wave* VI is called is the phase specified at the *phase in* control, or whether it is the phase available at the *phase out* control when the VI last executed. A TRUE value of *reset phase* sets the initial phase to *phase in*, whereas a FALSE value sets it to the value of *phase out* when the VI last executed.

The *wave* VIs are all reentrant (can keep track of phase internally) and accept frequency in normalized units (cycles/sample). The only *pattern* VI

that presently uses normalized units is the **Chirp Pattern** VI. Setting the *reset phase* Boolean to FALSE allows for continuous sampling simulation.

☞ **Note** ***Wave VIs are reentrant and accept the frequency input in terms of normalized units.***

In the next activity, you will generate a sine wave using both the **Sine Wave** VI and the **Sine Pattern** VI. You will see how in the **Sine Wave** VI you have more control over the initial phase than in the **Sine Pattern** VI.

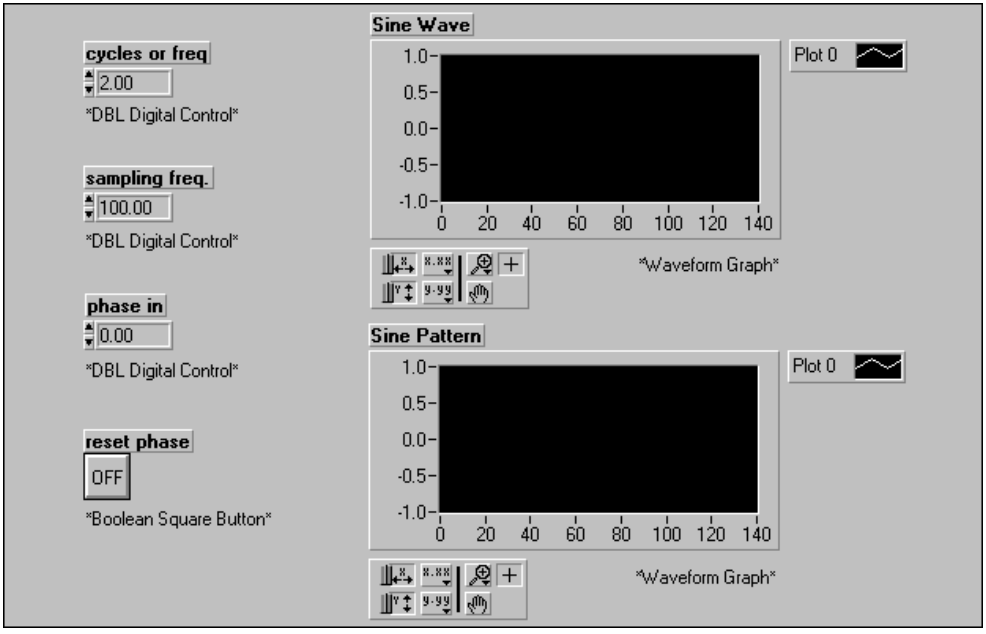


Activity 12-2. Use the Sine Wave and Sine Pattern VIs

Your objective is to generate a sinusoidal waveform using both the Sine Wave VI and the Sine Pattern VI and to understand the differences.

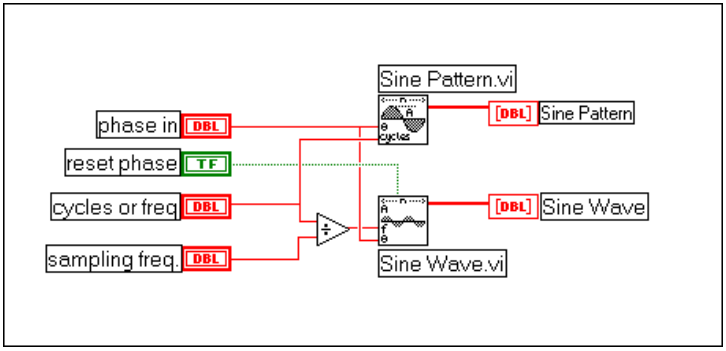
Front Panel

1. Open a new front panel and create the objects as shown in the following illustration.



Block Diagram

2. Build the block diagram shown in the following illustration.



Sine Pattern VI (**Analysis»Signal Generation** palette).

Sine Wave VI (**Analysis»Signal Generation** palette).

3. Save the VI as `Wave and Pattern.vi` in the `LabVIEW\Activity` directory.

4. Set the controls to the following values:

   cycles/freq: 2.00

   sampling freq: 100

   phase in: 0.00

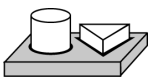
   reset phase: OFF

   Run the VI several times.

   Observe that the **Sine Wave** plot changes each time you run the VI. Because **reset phase** is set to OFF, the phase of the sine wave changes with each call to the VI, being equal to the value of **phase out** during the previous call. However, the Sine Pattern plot always remains the same, showing 2 cycles of the sinusoidal waveform. The initial phase of the Sine Pattern plot is equal to the value set in the **phase in** control.

Note ***"Phase in" and "phase out" are specified in degrees.***

5. Change **phase in** to 90 and run the VI several times. Just as before, the Sine Wave plot changes each time you run the VI. However, the Sine Pattern plot does not change, but the initial phase of the sinusoidal pattern is 90 degrees—the same as that specified in the **phase in** control.
6. With **phase in** still at 90, set **reset phase** to ON and run the VI several times. The sinusoidal waveforms shown in both the Sine Wave and Sine Pattern plots start at 90 degrees, but do not change with successive calls to the VI.
7. Keeping **reset phase** as ON, run the VI several times for each of the following values of **phase in**: 45, 180, 270, and 360. Note the initial phase of the generated waveform each time that the VI is run.



End of Activity 12-2.

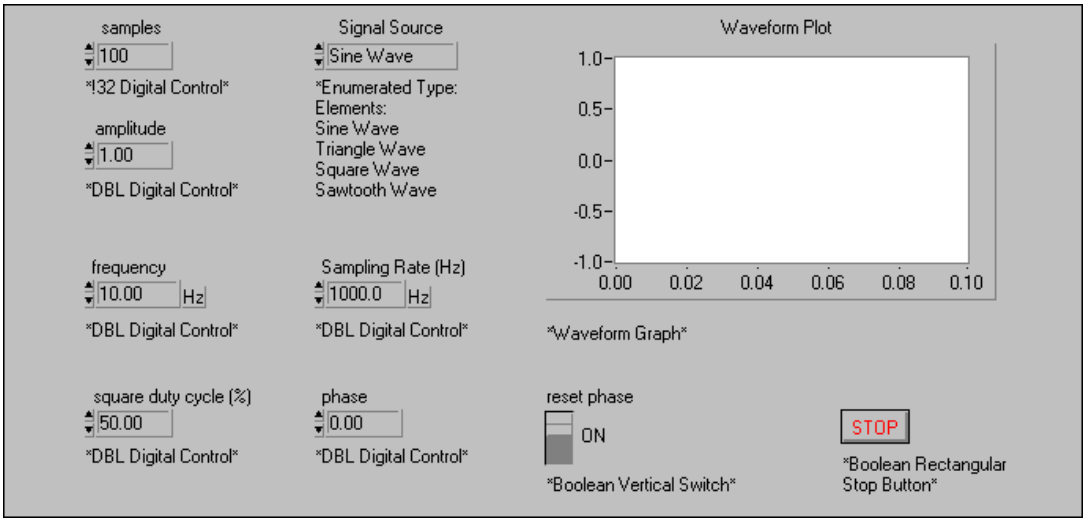
Activity 12-3. Build a Function Generator

Your objective is to build a simple function generator that can generate the following waveforms.

- ***Sine Wave***
- ***Square Wave***
- ***Triangle Wave***
- ***Sawtooth Wave***

Front Panel

1. Open a new front panel and create the objects as shown in the following illustration.



The **signal source** control selects the type of waveform that you want to generate.

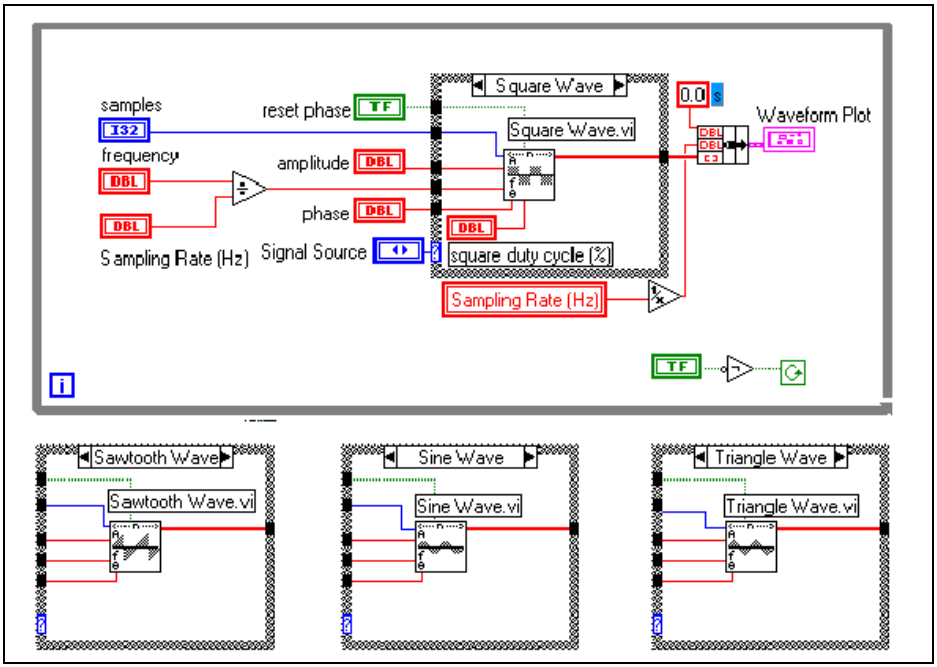
The **square duty cycle** control is used only for setting the duty cycle of the square wave.

The **samples** control determines the number of samples in the plot.

Notice that these are all *wave* VIs, and therefore they require the frequency input to be the normalized frequency. So, you divide **frequency** by the **sample rate** and the result is the normalized frequency wired to the *f* input of the VIs.

Block Diagram

2. Build the block diagram shown in the following illustration.



Sine Wave VI (**Analysis»Signal Generation** palette) generates a sine wave of normalized frequency *f*.

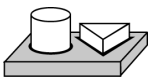
Triangle Wave VI (**Analysis»Signal Generation** palette) generates a triangular wave of normalized frequency *f*.

Square Wave VI (**Analysis»Signal Generation** palette) generates a square wave of normalized frequency *f* with specified duty cycle.

Sawtooth Wave VI (**Analysis»Signal Generation** palette) generates a sawtooth wave of normalized frequency *f*.

3. Save the VI as `Function Generator.vi` in the `LabVIEW\Activity` directory.
4. Select a **sampling rate** of 1000 Hz, **amplitude** = 1, **samples** = 100, **frequency** = 10, **reset phase** = ON, and **signal source** = sine wave. Because **sampling rate** = 1000 and **frequency** = 10 Hz, every 100 samples corresponds to one cycle.

5. Run the VI and observe the resulting plot.
6. Change **samples** to 200, 300, and 400. How many cycles of the waveform do you see? Explain why.
7. With **samples** set to 100, change **reset phase** to OFF. Do you notice any difference in the plot?
8. Change **frequency** to 10.01 Hz. What happens? Why?
9. Change **reset phase** to ON. Now what happens? Explain why.
10. Repeat steps 4 through 9 for different waveforms selected in the **signal source** control.



End of Activity 12-3.

13

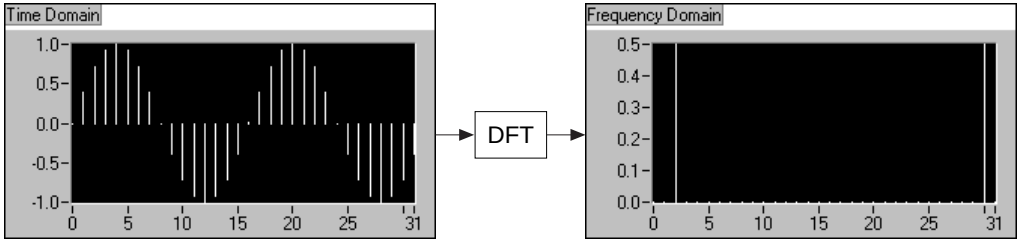
Digital Signal Processing

This chapter describes the fundamentals of the Fast Fourier Transform (FFT) and the Discrete Fourier Transform (DFT) and how they are used in spectral analysis. For examples of how to use the digital signal processing VIs, see the examples located in `examples\analysis\dspxmpl.llb`.

The Fast Fourier Transform (FFT)

The samples of a signal obtained from a DAQ board constitute the *time domain* representation of the signal. This representation gives the amplitudes of the signal at the instants of *time* during which it had been sampled. However, in many cases you want to know the frequency content of a signal rather than the amplitudes of the individual samples. The representation of a signal in terms of its individual frequency components is known as the *frequency domain* representation of the signal. The frequency domain representation could give more insight about the signal and the system from which it was generated.

The algorithm used to transform samples of the data from the time domain into the frequency domain is known as the *discrete Fourier transform* or DFT. The DFT establishes the relationship between the samples of a signal in the time domain and their representation in the frequency domain. The DFT is widely used in the fields of spectral analysis, applied mechanics, acoustics, medical imaging, numerical analysis, instrumentation, and telecommunications.



time domain representation of x[n] — frequency domain representation

Suppose you have obtained *N* samples of a signal from a DAQ board. If you apply the DFT to *N* samples of this time domain representation of the

signal, the result is also of length *N* samples, but the information it contains is of the frequency domain representation. The relationship between the *N* samples in the time domain and the *N* samples in the frequency domain is explained below.

If the signal is sampled at a sampling rate of f_s Hz, then the time interval between the samples (that is, the sampling interval) is Δt, where

$$\Delta t = \frac{1}{f_s}$$

The sample signals are denoted by x[i], $0 \le i \le N-1$ (that is, you have a total of *N* samples). When the discrete Fourier transform, given by

$$X_k = \sum_{i=0}^{N-1} x_i e^{-j2\pi ik/N} \qquad (13\text{-}1)$$

for

$$k = 0, 1, 2, \ldots, N-1$$

is applied to these *N* samples, the resulting output (*X*[k], 0 ð k ð N–1) is the frequency domain representation of x[i]. Notice that both the time domain *x* and the frequency domain *X* have a total of *N* samples. Analogous to the *time spacing* of Δt between the samples of *x* in the time domain, you have a frequency spacing of

$$\Delta f = \frac{f_s}{N} = \frac{1}{N\Delta t}$$

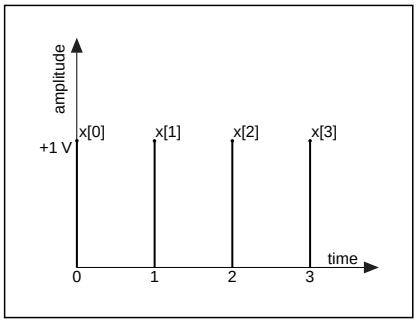
between the components of *X* in the frequency domain. Δf is also known as the *frequency resolution*. To increase the frequency resolution (smaller Δf) you must either increase the number of samples ***N*** (with *fs* constant) or decrease the sampling frequency *fs* (with *N* constant).

In the following example, you will go through the mathematics of Equation 13-1 to calculate the DFT for a D.C. signal.

DFT Calculation Example

In the next section, you will see the exact frequencies to which the *N* samples of the DFT correspond. For the present discussion, assume that *X*[0] corresponds to D.C., or the average value, of the signal. To see the

result of calculating the DFT of a waveform with the use of Equation 13-1, consider a D.C. signal having a constant amplitude of +1 V. Four samples of this signal are taken, as shown in the figure below.



Each of the samples has a value +1, giving the time sequence

$$x[0] = x[1] = x[3] = x[4] = 1$$

Using Equation 13-1 to calculate the DFT of this sequence and making use of Euler's identity,

$$\exp(-i\theta) = \cos(\theta) - j\sin(\theta)$$

you get:

$$X[0] = \sum_{i=0}^{N-1} x_i e^{-j2\pi i0/N} = x[0] + x[1] + x[2] + x[3] = 4$$

$$X[1] = x[0] + x[1]\left(\cos\left(\frac{\pi}{2}\right) - j\sin\left(\frac{\pi}{2}\right)\right) + x[2](\cos(\pi) - j\sin(\pi)) + x[3]\left(\cos\left(\frac{3\pi}{2}\right) - j\sin\left(\frac{3\pi}{2}\right)\right) = (1 - j - 1 + j) = 0$$

$$X[2] = x[0] + x[1](\cos(\pi) - j\sin(\pi)) + x[2](\cos(2\pi) - j\sin(2\pi)) + x[3](\cos(3\pi) - j\sin(3\pi)) = (1 - 1 + 1 - 1) = 0$$

$$X[3] = x[0] + x[1]\left(\cos\left(\frac{3\pi}{2}\right) - j\sin\left(\frac{3\pi}{2}\right)\right) + x[2](\cos(3\pi) - j\sin(3\pi)) + x[3]\left(\cos\left(\frac{9\pi}{2}\right) - j\sin\left(\frac{9\pi}{2}\right)\right) = (1 - j - 1 - j) = 0$$

Therefore, except for the DC component, *X*[0], all the other values are zero, which is as expected. However, the calculated value of *X*[0] depends on the value of *N* (the number of samples). Because you had *N* = 4, X[0] = 4. If *N* = 10, then you would have calculated *X*[0] = 10. This dependency of *X*[ ] on *N* also occurs for the other frequency components. Thus, you usually divide the DFT output by *N*, so as to obtain the correct magnitude of the frequency component.

Magnitude and Phase Information

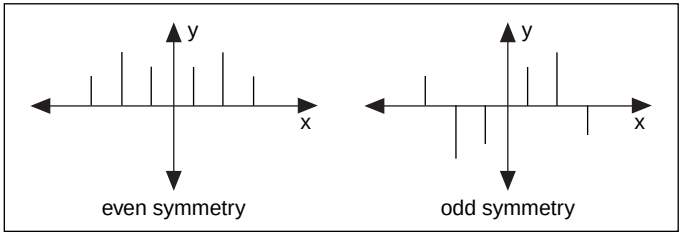
You have seen that *N* samples of the input signal result in *N* samples of the DFT. That is, the number of samples in both the time and frequency representations is the same. From Equation 13-1, you see that regardless of whether the input signal *x*[i] is real or complex, *X*[k] is always complex (although the imaginary part may be zero). Thus, because the DFT is complex, it contains two pieces of information—the amplitude and the phase. It turns out that for real signals (*x*[i] real) such as those obtained from the output of one channel of a DAQ board, the DFT is symmetric with the following properties:

$$| X[k] | = | X[N-k] |$$

and

$$\text{phase} (X[k]) = - \text{phase}(X[N-k])$$

The terms used to describe this symmetry are that the magnitude of X[k] is *even symmetric*, and phase(X[k]) is *odd symmetric*. An even symmetric signal is one that is symmetric about the y-axis, whereas an odd symmetric signal is symmetric about the origin. This is shown in the following figures.



The net effect of this symmetry is that there is repetition of information contained in the *N* samples of the DFT. Because of this repetition of information, only half of the samples of the DFT actually need to be computed or displayed, as the other half can be obtained from this repetition.

Note ***If the input signal is complex, the DFT will be nonsymmetric and you cannot use this trick.***

Frequency Spacing between DFT/FFT Samples

If the sampling interval is Δt seconds, and the first ($k = 0$) data sample is at 0 seconds, then the kth ($k > 0$, k integer) data sample is at $k\Delta t$ seconds. Similarly, if the frequency resolution is Δf *Hz*

($\Delta f = \frac{f_s}{N}$) then the kth sample of the DFT occurs at a frequency of $k\Delta f$ Hz. (Actually, as you will soon see, this is valid for only up to the first half of the frequency components. The other half represent negative frequency components.) Depending on whether the number of samples, N, is even or odd, you can have a different interpretation of the frequency corresponding to the k^{th} sample of the DFT.

For example, suppose N is even and let $p = \frac{N}{2}$. The following table shows the frequency to which each format element of the complex output sequence X corresponds.

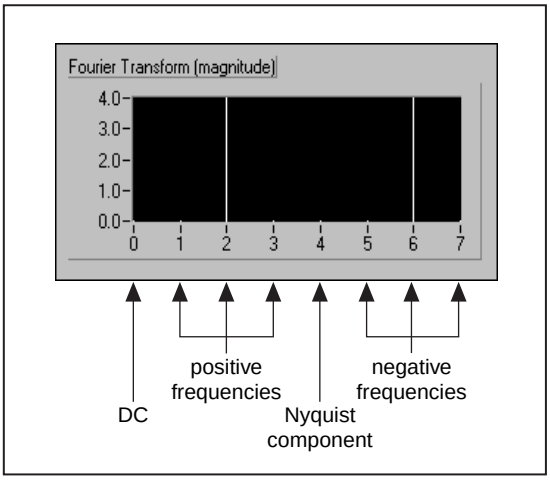
Note that the p^{th} element, X[p], corresponds to the Nyquist frequency. The negative entries in the second column beyond the Nyquist frequency represent *negative* frequencies.

For example, if $N = 8$, $p = N/2 = 4$, then

X[0]	DC
X[1]	Δf
X[2]	$2\Delta f$
X[3]	$3\Delta f$
X[4]	$4\Delta f$ (Nyquist freq)
X[5]	$-3\Delta f$
X[6]	$-2\Delta f$
X[7]	$-\Delta f$

Here, X[1] and X[7] will have the same magnitude, X[2] and X[6} will have the same magnitude, and X[3] and X[5] will have the same magnitude. The difference is that whereas X[1], X[2], and X[3] correspond to positive frequency components, X[5], X[6], and X[7] correspond to negative frequency components. Note that X[4] is at the Nyquist frequency.

The following illustration represents this complex sequence for $N = 8$.



Such a representation, where you see both the positive and negative frequencies, is known as the *two-sided* transform.

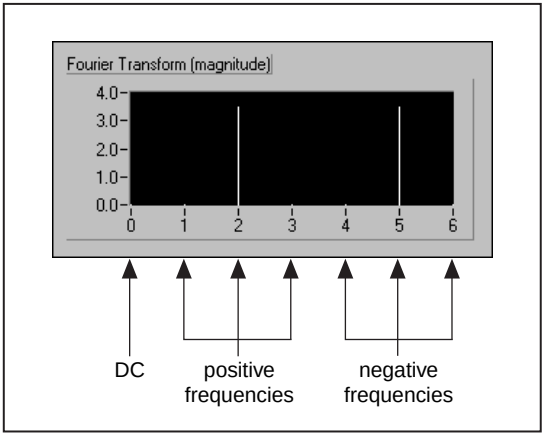
Note that when N is odd, there is no component at the Nyquist frequency.

For example, if N = 7, p = (N–1)/2 = (7–1)/2 = 3, and you have

X[0]	DC
X[1]	Δf
X[2]	$2\Delta f$
X[3]	$3\Delta f$
X[4]	$-3\Delta f$
X[5]	$-2\Delta f$
X[6]	$-\Delta f$

Now X[1] and X[6] have the same magnitude, X[2] and X[5] have the same magnitude, and X[3] and X[4] have the same magnitude. However, whereas X[1], X[2], and X[3] correspond to positive frequencies, X[4], X[5], and X[6] correspond to negative frequencies. Because N is odd, there is no component at the Nyquist frequency.

The following illustration represents the preceding table for $N = 7$.



This is also a two-sided transform, because you have both the positive and negative frequencies.

Fast Fourier Transforms

Direct implementation of the DFT on N data samples requires approximately N^2 complex operations and is a time-consuming process. However, when the size of the sequence is a power of 2,

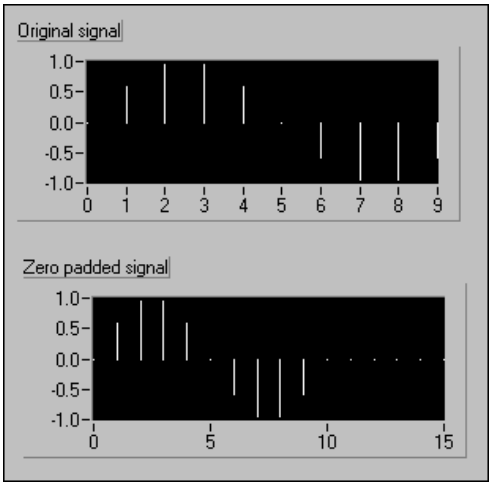
$$N = 2^m \quad \text{for } m = 1, 2, 3,\ldots$$

you can implement the computation of the DFT with approximately $N \log_2(N)$ operations. This makes the calculation of the DFT much faster, and DSP literature refers to these algorithms as fast Fourier transforms (FFTs). The FFT is nothing but a fast algorithm for calculating the DFT when the number of samples (N) is a power of 2.

The advantages of the FFT include speed and memory efficiency, because the VI can compute the FFT “in place,” that is, no additional memory buffers are needed to compute the output. The size of the input sequence, however, must be a power of 2. The DFT can efficiently process any size sequence, but the DFT is slower than the FFT and uses more memory, because it must allocate additional buffers for storing intermediate results during processing.

Zero Padding

A technique employed to make the input sequence size equal to a power of 2 is to add zeros to the end of the sequence so that the total number of samples is equal to the next higher power of 2. For example, if you have 10 samples of a signal, you can add six zeros to make the total number of samples equal to 16 (= 2^4—a power of 2). This is shown below:



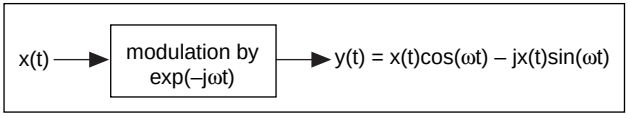
The addition of zeros to the end of the time domain waveform does not affect the spectrum of the signal. In addition to making the total number of samples a power of two so that faster computation is made possible by using the FFT, zero padding also helps in increasing the frequency resolution (recall that Δf = fs/N) by increasing the number of samples, *N*.

FFT VIs in the Analysis Library

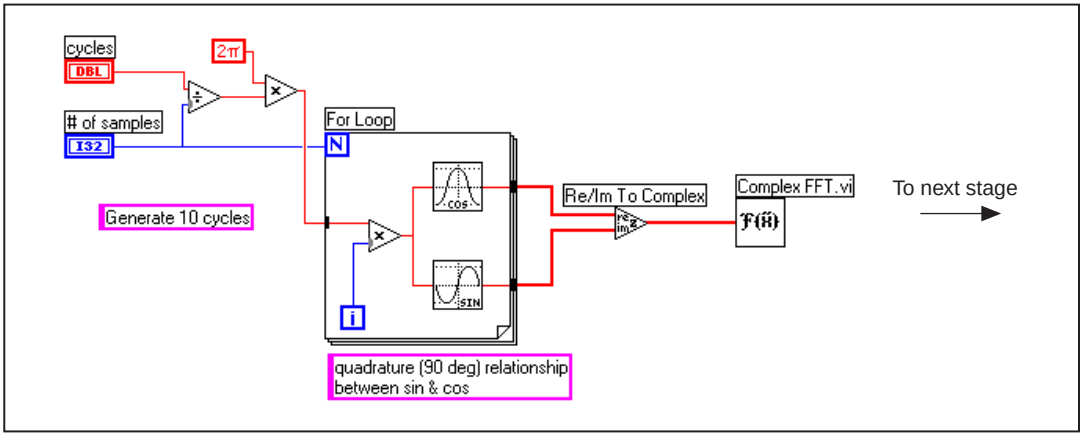
The analysis library contains two VIs that compute the FFT of a signal. They are the **Real FFT** VI and **Complex FFT** VI.

The difference between the two VIs is that the **Real FFT** VI computes the FFT of a real-valued signal, whereas the **Complex FFT** VI computes the FFT of a complex-valued signal. However, keep in mind that the outputs of both VIs are complex.

Most real-world signals are real valued, and hence you can use the **Real FFT** VI for most applications. Of course, you could also use the **Complex FFT** VI by setting the imaginary part of the signal to zero. An example of an application where you could use the **Complex FFT** VI is when the signal consists of both a real and imaginary component. Such a type of signal occurs frequently in the field of telecommunications, where you modulate a waveform by a complex exponential. The process of modulation by a complex exponential results in a complex signal, as shown below:



The block diagram below shows a simplified version of how you can generate 10 cycles of a complex signal:

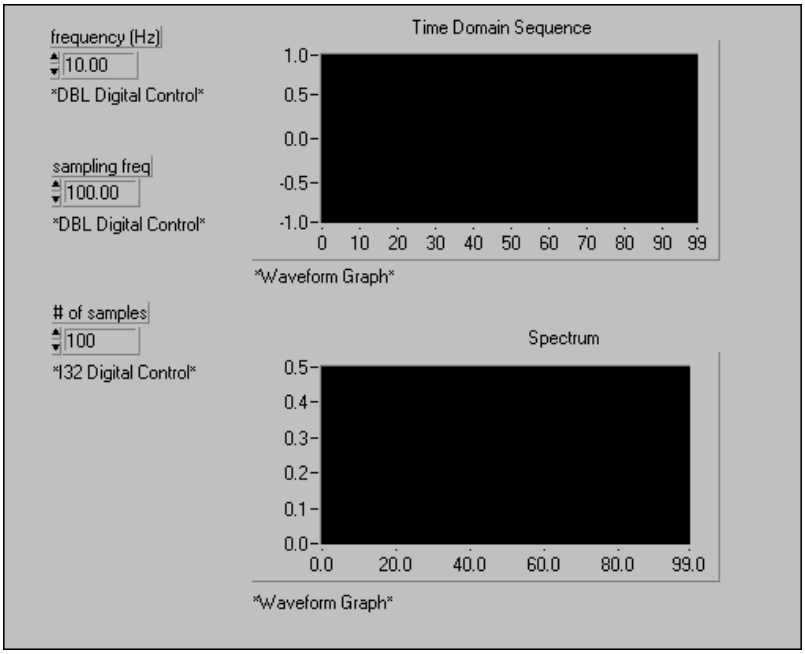


Activity 13-1. Use the Real FFT VI

In this activity, your objective is to display the two-sided and the one-sided Fourier transform of a signal using the Real FFT VI, and to observe the effect of aliasing in the frequency spectrum.

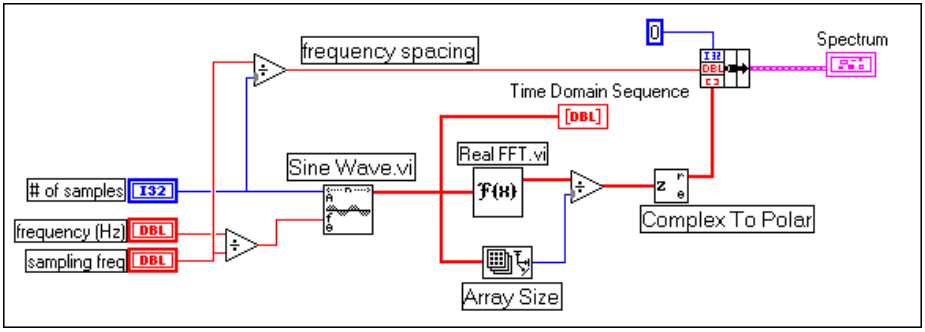
Front Panel

1. Build the front panel shown in the following illustration.



Block Diagram

2. Build the block diagram shown in the following illustration.



Array Size function (**Functions»Array** palette) scales the output of the FFT by the **number of samples** so as to obtain the correct amplitude of the frequency components.

Sine Wave function (**Functions»Analysis»Signal Generation** palette) generates a time domain sinusoidal waveform.

Real FFT function (**Functions»Analysis»Digital Signal Processing** palette) computes the FFT of the input data samples.

Complex to Polar function (**Functions»Numeric»Complex** palette) separates the complex output of the FFT into its real and imaginary (magnitude and phase) parts. The phase information is in units of radians. Here you are displaying only the magnitude of the FFT.

The frequency spacing, Δf, is given by dividing the **sampling freq** by the **# of samples**.

3. Save this VI as `FFT_2sided.vi` in the `LabVIEW\Activity` directory.
4. Select **frequency (Hz)** = 10, **sampling freq** = 100, and **# of samples** = 100. Run the VI.

   Notice the plots of the time waveform and the frequency spectrum. Because **sampling freq** = **# of samples** = 100, you are in effect sampling for 1 second. Thus, the number of cycles of the sine wave you see in the time waveform is equal to the **frequency(Hz)** you select. In this case, you will see 10 cycles. (If you change the **frequency (Hz)** to 5, you will see five cycles).

Two-Sided FFT

5. Examine the frequency spectrum (the Fourier transform). You will notice two peaks, one at 10 Hz and the other at 90 Hz. The peak at 90 Hz is actually the negative frequency of 10 Hz. The plot you see is known as the *2-sided FFT* because it shows both the positive and the negative frequencies.
6. Run the VI with **frequency (Hz)** = 10 and then with **frequency (Hz)** = 20. For each case, note the shift in both peaks of the spectrum.

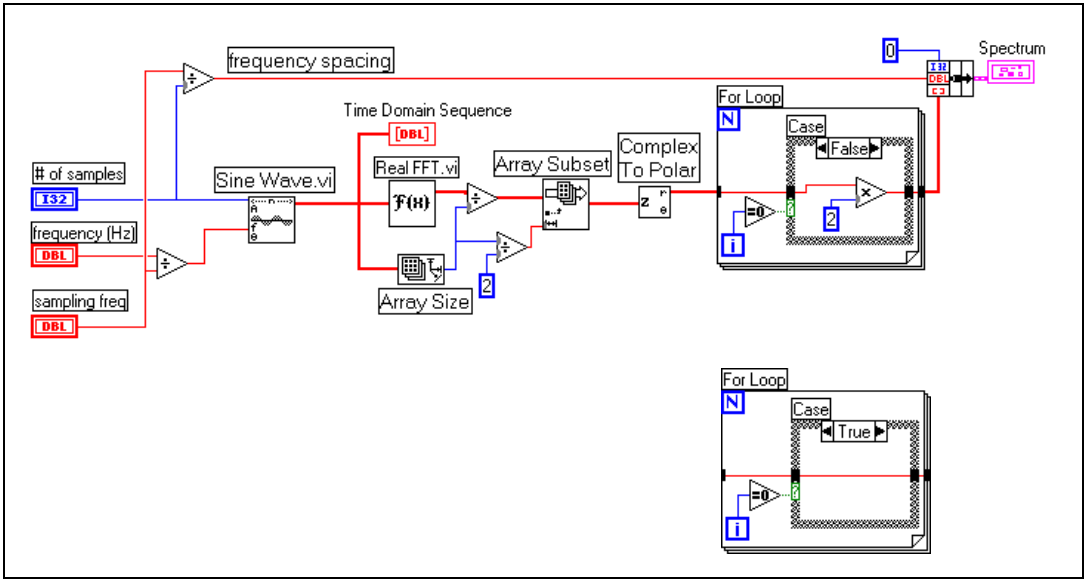
Note ***Also observe the time domain plot for frequency (Hz) = 10 and 20. Which one gives a better representation of the sine wave? Why?***

7. Because fs = 100 Hz, you can accurately sample only signals having a frequency < 50 Hz (Nyquist frequency = fs/2). Change the **frequency (Hz)** to 48 Hz. You should see the peaks at ± 48 Hz on the spectrum plot.
8. Now change the **frequency (Hz)** to 52 Hz. Is there any difference between the result of step 5 and what you see on the plots now? Because 52 > Nyquist, the frequency of 52 is aliased to |100 – 52| = 48 Hz.
9. Change **frequency (Hz)** to 30 Hz and 70 Hz and run the VI. Is there any difference between the two cases? Explain why.

One-Sided FFT

10. Modify the block diagram of the VI as shown below. You have seen that the FFT had repetition of information because it contained information about both the positive and the negative frequencies. This modification now shows only half the FFT points (only the positive frequency components). This representation is known as the *1-sided FFT*. The 1-sided FFT shows only the positive frequency components. Note that you need to multiply the positive frequency

components by two to obtain the correct amplitude. The D.C. component, however, is left untouched.



Equal To O? function (**Functions»Comparison** palette) tests to see if the array index is equal zero. If so, it corresponds to the D.C. component and should not be multiplied by two.

11. Run the VI with the following values: **frequency (Hz)** = 30, **sampling freq** = 100, **# of samples** = 100.
12. Save the VI as `FFT_1sided.vi` in the `LabVIEW\Activity` directory.
13. Change the value of **frequency (Hz)** to 70 and run the VI. Do you notice any difference between this and the result of step 9?

End of Activity 13-1.

The Power Spectrum

You have seen that the DFT (or FFT) of a real signal is a complex number, having a real and an imaginary part. The *power* in each frequency component represented by the DFT/FFT can be obtained by squaring the magnitude of that frequency component. Thus, the power in the k^{th} frequency component (the k^{th} element of the DFT/FFT) is given by $|X[k]|^2$. The plot showing the power in each of the frequency components is known as the *power spectrum*. Because the DFT/FFT of a real signal is symmetric, the power at a positive frequency of $k\Delta f$ is the same as the power at the corresponding negative frequency of $-k\Delta f$ (DC and Nyquist components not included). The total power in the DC and Nyquist components are $|X[0]|^2$ and $\left|X\left[\frac{N}{2}\right]\right|^2$, respectively.

Loss of Phase Information

Because the power is obtained by squaring the magnitude of the DFT/FFT, the power spectrum is always real. The disadvantage of this is that the phase information is lost. If you want phase information, you must use the DFT/FFT, which gives you a complex output.

You can use the power spectrum in applications where phase information is not necessary (for example, to calculate the harmonic power in a signal). You can apply a sinusoidal input to a nonlinear system and see the power in the harmonics at the system output.

Frequency Spacing between Samples

You can use the **Power Spectrum VI** in the **Analysis»Digital Signal Processing** subpalette to calculate the power spectrum of the time domain data samples. Just like the DFT/FFT, the number of samples from the **Power Spectrum** VI output is the same as the number of data samples applied at the input. Also, the frequency spacing between the output samples is Δf = fs/N.

Summary

The time domain representation (sample values) of a signal can be converted into the frequency domain representation by means of an algorithm known as the discrete Fourier transform (DFT). To have fast calculation of the DFT, an algorithm known as the fast Fourier transform (FFT) is used. You can use this algorithm when the number of signal samples is a power of two.

The output of the conventional DFT/FFT is two-sided because it contains information about both the positive and the negative frequencies. This output can be converted into a one-sided DFT/FFT by using only half the DFT/FFT output points. The frequency spacing between the samples of the DFT/FFT is $\Delta f = fs/N$.

The power spectrum can be calculated from the DFT/FFT by squaring the magnitude of the individual frequency components. The **Power Spectrum** VI in the advanced analysis library does this automatically for you. The units of the output of the Power Spectrum VI are in $Vrms^2$. However, the power spectrum does not provide any phase information.

The DFT, FFT, and power spectrum are useful for measuring the frequency content of stationary or transient signals. The FFT provides the average frequency content of the signal over the entire time that the signal was acquired. For this reason, you use the FFT mostly for stationary signal analysis (when the signal is not significantly changing in frequency content over the time that the signal is acquired), or when you want only the average energy at each frequency line. A large class of measurement problems fall in this category. For measuring frequency information that changes during the acquisition, you should use the joint time-frequency analysis (JTFA) toolkit or the wavelet and filter banks designer (WFBD) toolkit.

14

Smoothing Windows

This chapter explains how using windows prevents spectral leakage and improves the analysis of acquired signals. For examples of how to use the analysis window VIs, see the examples located in `examples\analysis\windxmpl.llb`.

Introduction to Smoothing Windows

In practical signal-sampling applications, you can obtain only a finite record of the signal, even when you carefully observe the sampling theorem and sampling conditions. Unfortunately for the discrete-time system, the finite sampling record results in a truncated waveform that has different spectral characteristics from the original continuous-time signal. These discontinuities produce leakage of spectral information, resulting in a discrete-time spectrum that is a smeared version of the original continuous-time spectrum.

A simple way to improve the spectral characteristics of a sampled signal is to apply smoothing windows. When performing Fourier or spectral analysis on finite-length data, you can use windows to minimize the transition edges of your truncated waveforms, thus reducing spectral leakage. When used in this manner, smoothing windows act like predefined, narrowband, lowpass filters.

About Spectral Leakage and Smoothing Windows

When you use the DFT/FFT to find the frequency content of a signal, it is inherently assumed that the data that you have is a single period of a periodically repeating waveform. This is shown in Figure 14-1. The first period shown is the one sampled. The waveform corresponding to this period is then repeated in time to produce the periodic waveform.

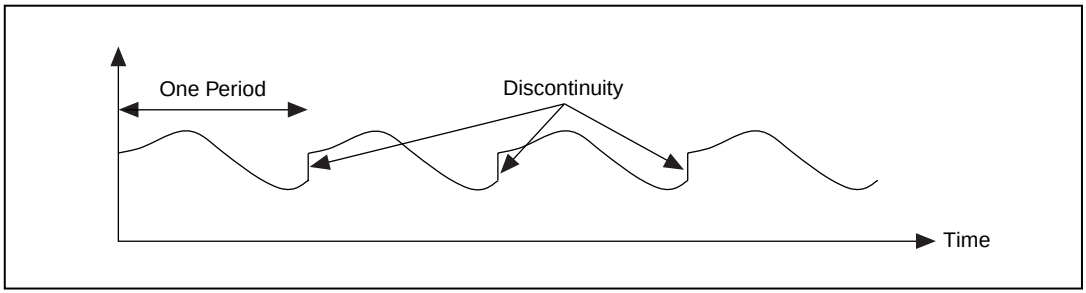


Figure 14-1. Periodic Waveform Created from Sampled Period

As seen in the previous figure, because of the assumption of periodicity of the waveform, discontinuities between successive periods will occur. This happens when you sample a noninteger number of cycles. These *artificial* discontinuities turn up as very high frequencies in the spectrum of the signal, frequencies that were not present in the original signal. These frequencies could be much higher than the Nyquist frequency, and as you have seen before, will be aliased somewhere between 0 and fs/2. The spectrum you get by using the DFT/FFT therefore will not be the actual spectrum of the original signal, but will be a smeared version. It appears as if the energy at one frequency has *leaked out* into all the other frequencies. This phenomenon is known as *spectral leakage*.

Figure 14-2 shows a sine wave and its corresponding Fourier transform. The sampled time domain waveform is shown in Graph 1. Because the Fourier transform assumes periodicity, you repeat this waveform in time, and the periodic time waveform of the sine wave of Graph 1 is shown in Graph 2. The corresponding spectral representation is shown in Graph 3. Because the time record in Graph 2 is periodic, with no discontinuities, its spectrum is a single line showing the frequency of the sine wave. The reason that the waveform in Graph 2 does not have any discontinuities is because you have sampled an integer number of cycles (in this case, 1) of the time waveform.

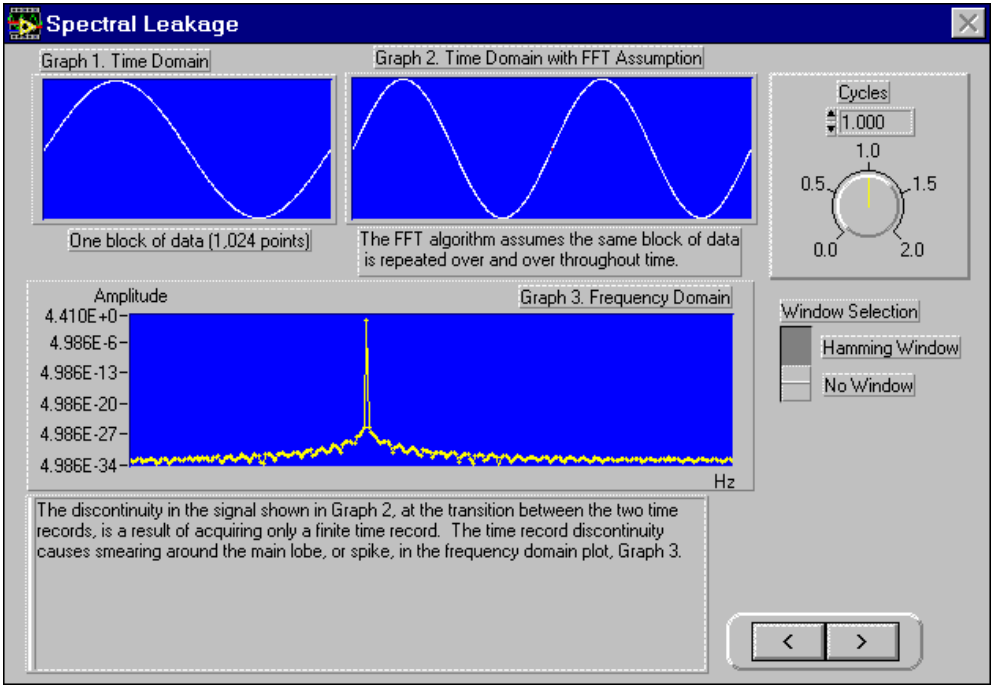


Figure 14-2. Sine Wave and Corresponding Fourier Transform

In Figure 14-3, you see the spectral representation when you sample a noninteger number of cycles of the time waveform (namely 1.25). Graph 1 now consists of 1.25 cycles of the sine wave. When you repeat this periodically, the resulting waveform, as shown in Graph 2, consists of discontinuities. The corresponding spectrum is shown in Graph 3. Notice how the energy is now spread over a wide range of frequencies. This smearing of the energy is *spectral leakage*. The energy has leaked out of one of the FFT lines and smeared itself into all the other lines.

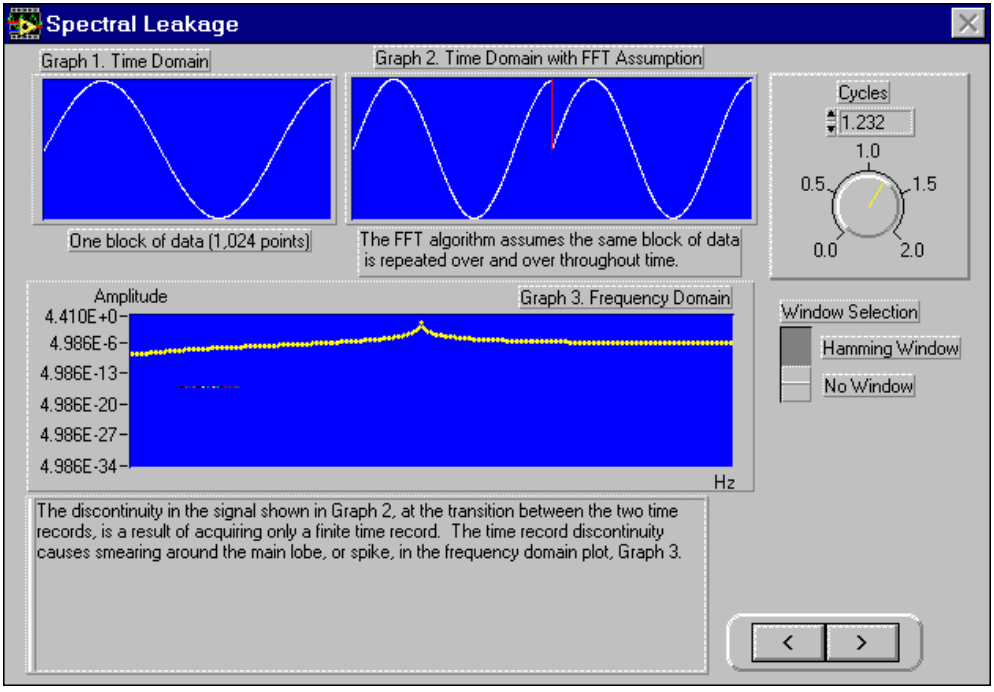


Figure 14-3. Spectral Representation When Sampling a Nonintegral Number of Samples

Leakage exists because of the finite time record of the input signal. To overcome leakage, one solution is to take an infinite time record, from –infinity to +infinity. Then the FFT would calculate one single line at the correct frequency. Waiting for infinite time is, however, not possible in practice. So, because you are limited to having a finite time record, another technique, known as *windowing*, is used to reduce the spectral leakage.

The amount of spectral leakage depends on the amplitude of the discontinuity. The larger the discontinuity, the more the leakage, and vice versa. You can use windowing to reduce the amplitude of the discontinuities at the boundaries of each period. It consists of multiplying the time record by a finite length window whose amplitude varies smoothly and gradually towards zero at the edges. This is shown in Figure 14-4, where the original time signal is windowed using a *Hanning* window. Notice that the time waveform of the windowed signal gradually tapers to zero at the ends. Therefore, when performing Fourier or spectral analysis on finite-length data, you can use windows to minimize the transition edges of your sampled waveform. A smoothing window function applied to the data before it is transformed into the frequency domain minimizes spectral leakage.

Note that if the time record contains an integral number of cycles, as shown in Figure 14-2, then the assumption of periodicity does not result in any discontinuities, and thus there is no spectral leakage. The problem arises only when you have a nonintegral number of cycles.

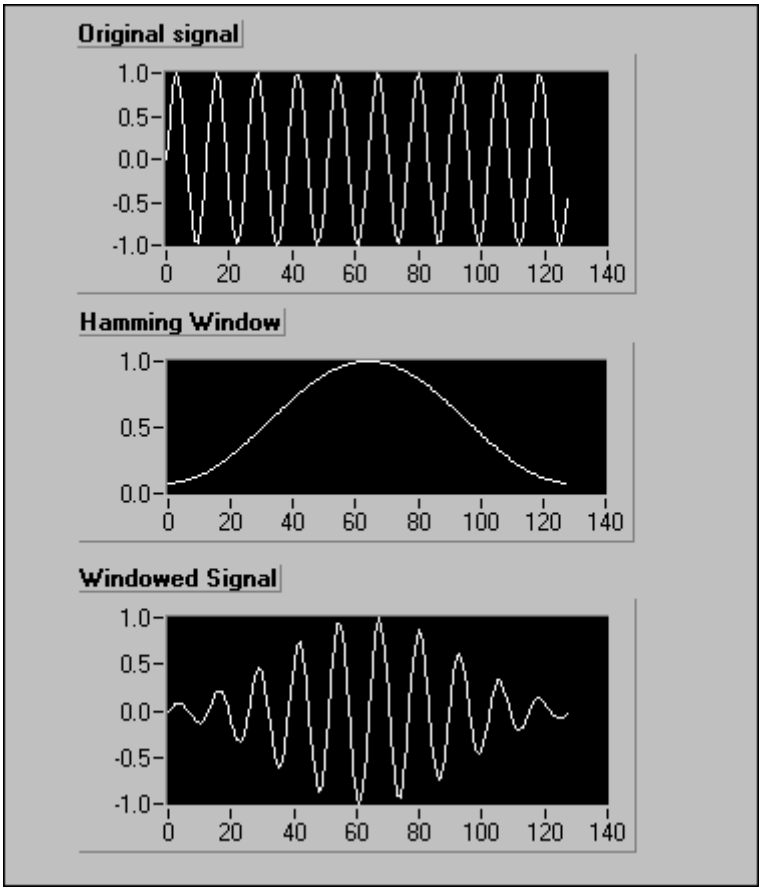


Figure 14-4. Time Signal Windowed Using a Hamming Window

Windowing Applications

There are several reasons to use windowing. Some of these are:

- To define the duration of the observation.
- Reduction of spectral leakage.
- Separation of a small amplitude signal from a larger amplitude signal with frequencies very close to each other.

Characteristics of Different Types of Window Functions

Applying a window to (windowing) a signal in the time domain is equivalent to multiplying the signal by the window function. Because multiplication in the time domain is equivalent to convolution in the frequency domain, the spectrum of the windowed signal is a convolution of the spectrum of the original signal with the spectrum of the window. Thus, windowing changes the shape of the signal in the time domain, as well as affecting the spectrum that you see.

Many different types of windows are available in the LabVIEW analysis library. Depending on your application, one may be more useful than the others. Some of these windows are:

Rectangular (None)

The rectangular window has a value of one over its time interval. Mathematically, it can be written as:

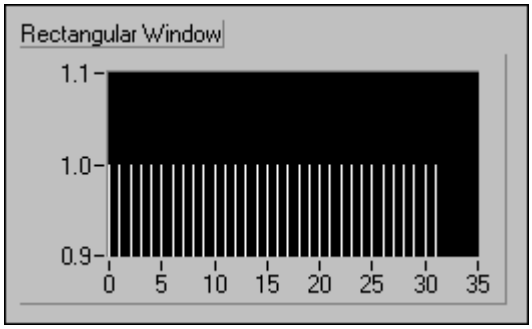
$$w[n] = 1.0$$

for

$$n = 0, 1, 2.....N-1$$

where *N* is the length of the window. Applying a rectangular window is equivalent to not using any window. This is because the rectangular function just truncates the signal to within a finite time interval. The rectangular window has the highest amount of spectral leakage.

The rectangular window for $N = 32$ is shown in the following illustration:



The rectangular window is useful for analyzing transients that have a duration shorter than that of the window. It is also used in *order tracking*, where the effective sampling rate is proportional to the speed of the shaft in rotating machines. In this application, it detects the main mode of vibration of the machine and its harmonics.

Hanning

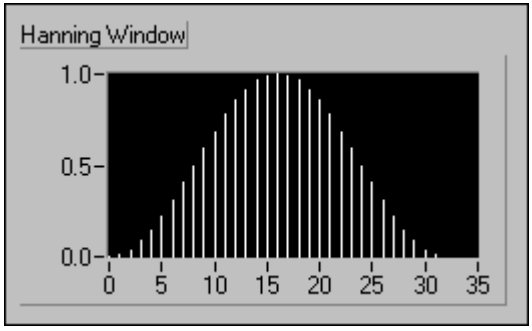
This window has a shape similar to that of half a cycle of a cosine wave. Its defining equation is

$$w(n) = 0.5 - 0.5\cos(2\pi n/N)$$

for

$$n = 0, 1, 2, ...N-1$$

A Hanning window with N = 32 is shown below:



The Hanning window is useful for analyzing transients longer than the time duration of the window, and also for general purpose applications.

Hamming

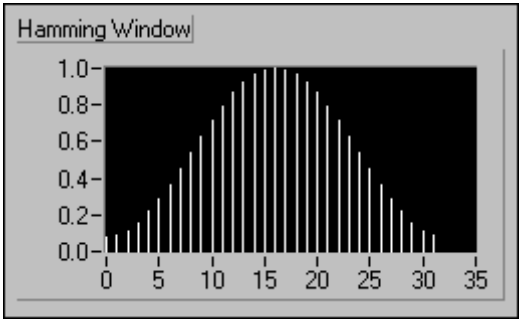
This window is a modified version of the Hanning window. Its shape is also similar to that of a cosine wave. It can be defined as

$$w(n) = 0.54 - 0.46\cos(2\pi n/N)$$

for

$$n = 0, 1, 2, ...N-1$$

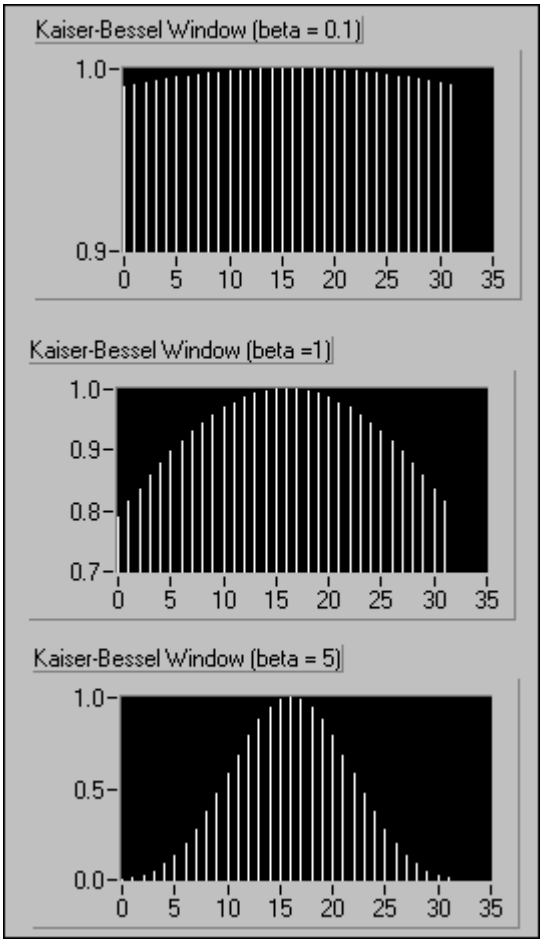
A Hamming window with N = 32 is shown below:



You see that the Hanning and Hamming windows are somewhat similar. However, note that in the time domain, the Hamming window does not get as close to zero near the edges as does the Hanning window.

Kaiser-Bessel

This window is a “flexible” window whose shape the user can modify by adjusting the parameter *beta*. Thus, depending on your application, you can change the shape of the window to control the amount of spectral leakage. The Kaiser-Bessel window for different values of *beta* are shown below:



Note that for small values of beta, the shape is close to that of a rectangular window. Actually, for beta = 0 .0, you do get a rectangular window. As you increase beta, the window tapers off more to the sides.

This window is good for detecting two signals of almost the same frequency, but significantly different amplitudes.

Triangle

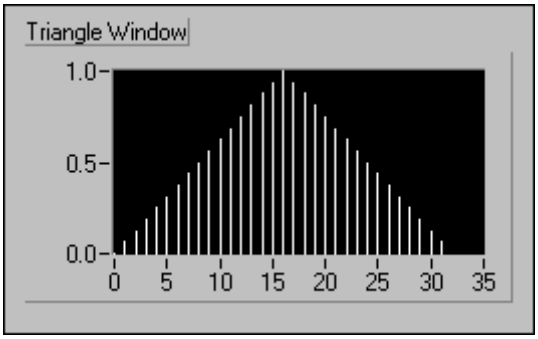
The shape of this window is that of a triangle. It is given by

$$w[n] = 1 - | (2n-N) / N |$$

for

$$n = 0, 1, 2, ..., n-1$$

A triangle window for N = 32 is shown below:



Flattop

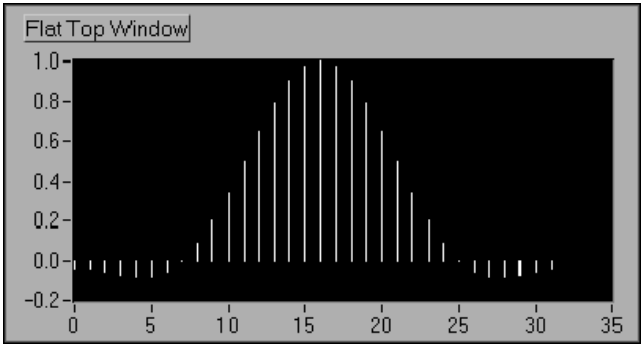
This window has the best amplitude accuracy of all the window functions. The increased amplitude accuracy (± 0.02 dB for signals exactly between integral cycles) is at the expense of frequency selectivity. The Flattop window is most useful in accurately measuring the amplitude of single frequency components with little nearby spectral energy in the signal. The Flattop window can be defined as

$$w(n) = a0 - a1*\cos(2\pi n/N) + a2*\cos(4\pi n/N)]$$

where

$$a0 = 0.2810638602$$
$$a1 = 0.5208971735$$
$$a2 = 0.1980389663$$

A flattop window is shown below:



Exponential

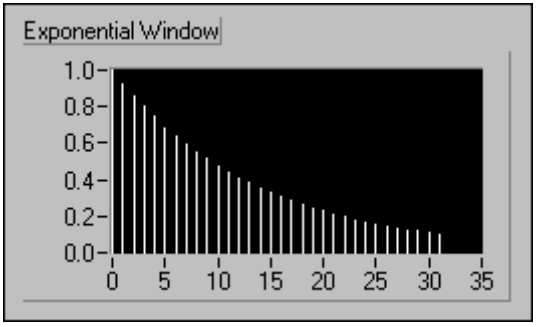
The shape of this window is that of a decaying exponential. It can be mathematically expressed as:

$$w[n] = e^{\left(\frac{n\ln(f)}{N-1}\right)} = f^{\left(\frac{n}{N-1}\right)}$$

for

$$n = 0, 1, 2.....N-1$$

where f is the final value. The initial value of the window is one, and it gradually decays towards zero. The final value of the exponential can be adjusted to between 0 and 1. The exponential window for N = 32, with the final value specified as 0.1, is shown below:



This window is useful in analyzing transients (signals that exist only for a short time duration) whose duration is longer than the length of the window.

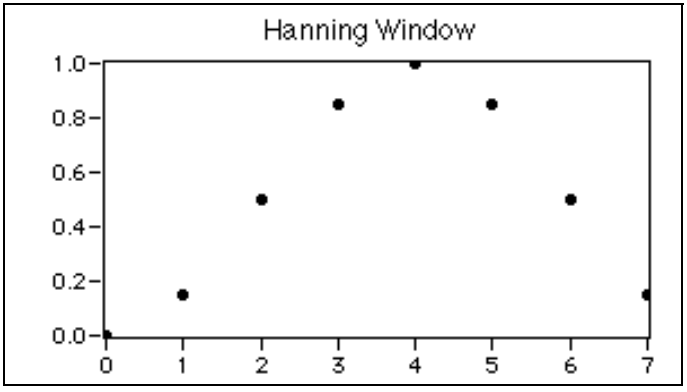
This window can be applied to signals that decay exponentially, such as the response of structures with light damping that are excited by an impact (for example, a hammer).

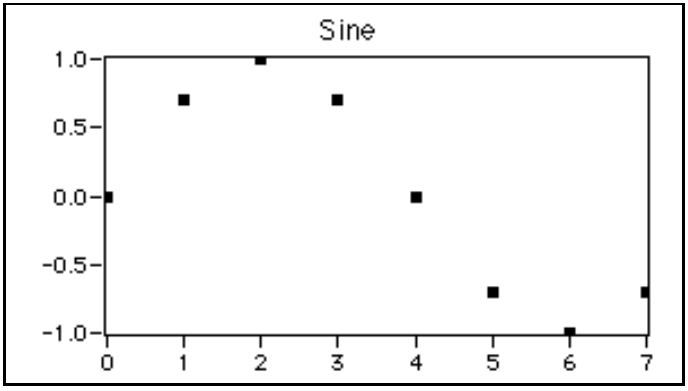
Windows for Spectral Analysis Versus Windows for Coefficient Design

The window VIs implemented in the Analysis library in LabVIEW are designed for spectral analysis applications. In these applications, the input signal is windowed by passing it through one of the window VIs. The windowed signal is then passed to a DFT-based VI for frequency-domain display and analysis.

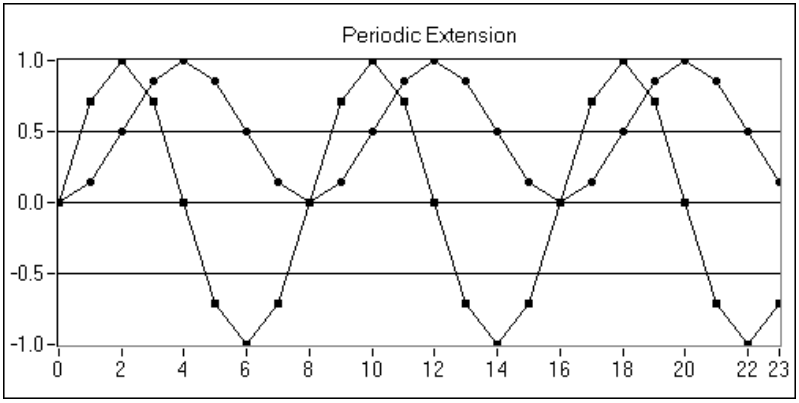
The window functions designed for spectral analysis must be *DFT-even*, a term defined by Fredric J. Harris in his paper *On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform* (*Proceedings of the IEEE*, Volume 66, No.1, January 1978). A window function is DFT-even if its dot product (inner product) with integral cycles of sine sequences is identically zero. Another way to think of a DFT-even sequence is that its DFT has no imaginary component.

The following figures illustrate the Hanning window and one cycle of a sine pattern for a sample size of 8. The figures below show that the DFT-even Hanning window is not symmetric about its midpoint and its last point is not equal to its first point, much like one complete cycle of a sine pattern.





Finally, the DFT considers input sequences to be periodic—that the signal being analyzed is actually a concatenation of the input signal. The following illustration shows three such cycles of the previous sequences, demonstrating the smooth periodic extension of the DFT-even window and the single-cycle sine pattern.



Another type of window application is that of FIR filter design, see the *Windowed FIR Filters* section in Chapter 16, *Filtering*. This application requires windows that are symmetric about their midpoint.

The following equations of the Hanning window function illustrate the difference between the DFT-even window function (spectral analysis) and the symmetrical window function (coefficient design).

Hanning window function for spectral analysis:

$$w[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N}\right)\right)$$

for

$$i=0,1, 2, ..., N-1$$

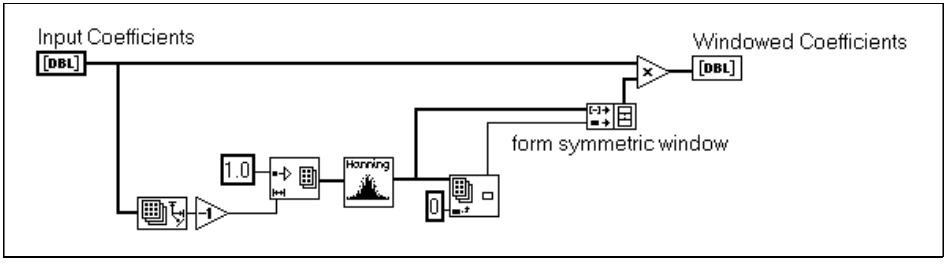
Hanning window function for symmetrical coefficient design:

$$w[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N-1}\right)\right)$$

for

$$i=0, 1, 2, ..., N-1$$

The two equations above show that you can implement the symmetrical window functions by slightly modifying the use of the DFT-even window functions. The following illustration shows a block diagram that uses the Hanning Window VI to implement symmetrical windowing of filter coefficients.



See Appendix A, *Analysis References*, for more information on smoothing windows.

What Type of Window Do I Use?

Now that you have seen several of the many different types of windows that are available, you may ask, “What type of window should I use?” The answer depends on the type of signal you have and what you are looking for. Choosing the correct window requires some prior knowledge of the signal that you are analyzing. In summary, the following table shows the different types of signals and the appropriate windows that you can use with them.

Type of signal	Window
Transients whose duration is shorter than the length of the window	Rectangular
Transients whose duration is longer than the length of the window	Exponential, Hanning
General-purpose applications	Hanning
Order tracking	Rectangular
System analysis (frequency response measurements)	Hanning (for random excitation), Rectangular (for pseudorandom excitation)
Separation of two tones with frequencies very close to each other, but with widely differing amplitudes	Kaiser-Bessel
Separation of two tones with frequencies very close to each other, but with almost equal amplitudes	Rectangular
Accurate single tone amplitude measurements	Flat Top

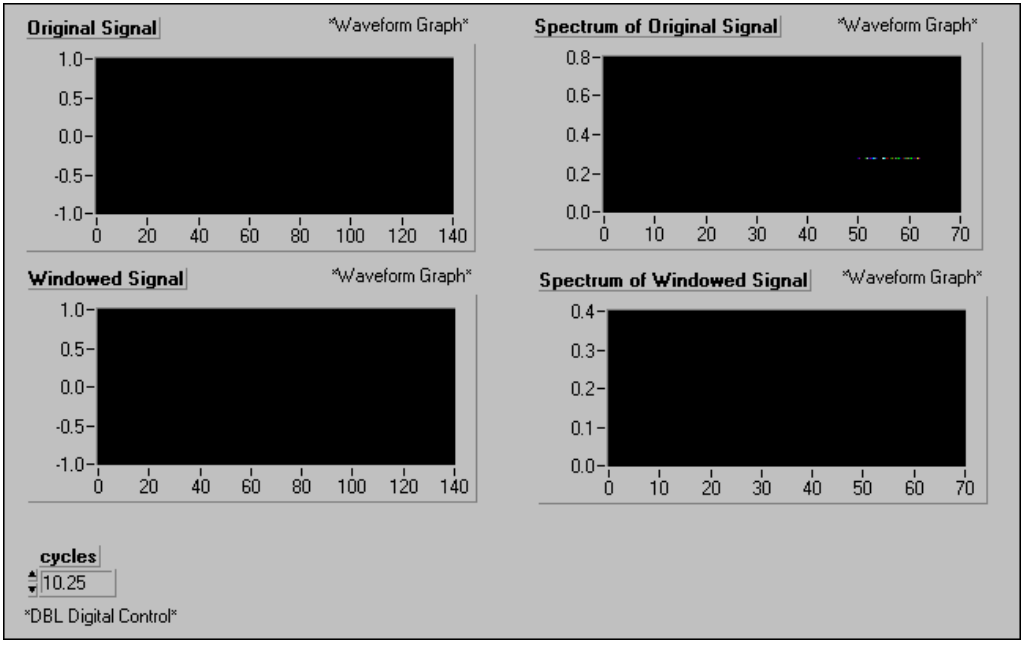
In many cases, you may not have sufficient prior knowledge of the signal, so you need to experiment with different windows to find the best one.

Activity 14-1. Compare a Windowed and Nonwindowed Signal

Your objective is to observe the difference (both time and frequency domains) between a windowed and nonwindowed signal.

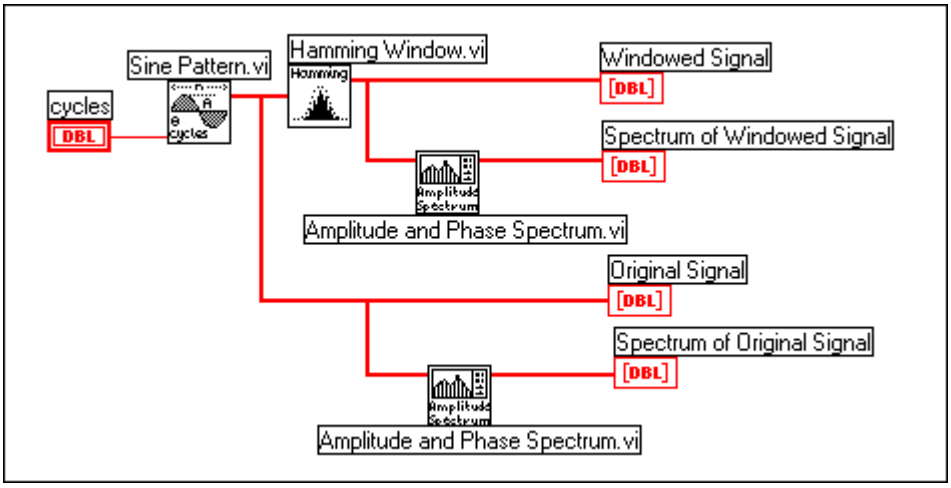
Front Panel

1. Open a new front panel and create the objects as shown in the following illustration.



Block Diagram

2. Build the block diagram shown in the following illustration.



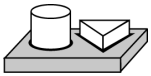
The Sine Pattern VI (**Functions»Analysis»Signal Generation** palette) generates a sine wave with the number of cycles specified in the **cycles** control.

The time waveform of the sine wave is windowed using the Hamming Window VI (**Functions»Analysis»Windows** palette), and both the windowed and nonwindowed time waveforms are displayed on the left two plots on the front panel.

The Amplitude and Phase Spectrum VI (**Functions»Analysis» Measurement** palette) obtains the amplitude spectrum of the windowed and nonwindowed time waveforms. These waveforms are displayed on the two plots on the right side of the front panel.

3. Save the VI as `Windowed & Unwindowed Signal.vi` in the `LabVIEW\Activity` directory.
4. Set **cycles** to 10 (an integral number) and run the VI. Note that the spectrum of the windowed signal is broader (wider) than the spectrum of the nonwindowed signal. But both the spectra are concentrated near 10 on the x-axis.

5. Change **cycles** to 10.25 (a nonintegral number) and run the VI. Note that the spectrum of the nonwindowed signal is now more spread out than it was before. This is because now you have a noninteger number of cycles, and when you repeat the waveform to make it periodic, you get discontinuities. The spectrum of the windowed signal is still concentrated, but that of the nonwindowed signal has now smeared all over the frequency domain. (This is spectral leakage.)
6. Change **cycles** to 10.5 and observe the frequency domain plots. Spectral leakage of the original signal is clearly apparent.



End of Activity 14-1.

15

Spectrum Analysis and Measurement

This chapter shows how to determine the amplitude and phase spectrum, develop a spectrum analyzer, and determine the total harmonic distortion (THD) of your signals. For examples of how to use the measurement VIs, see the examples located in `examples\analysis\measure\measxmpl.llb`.

Introduction to Measurement VIs

Several measurement VIs perform commonly used time domain to frequency-domain transformations such as amplitude and phase spectrum, signal power spectrum, network transfer function, and so on. Other measurement VIs interact with VIs that perform such functions as scaled time domain windowing and power and frequency estimation.

You can use the measurement VIs for the following applications:

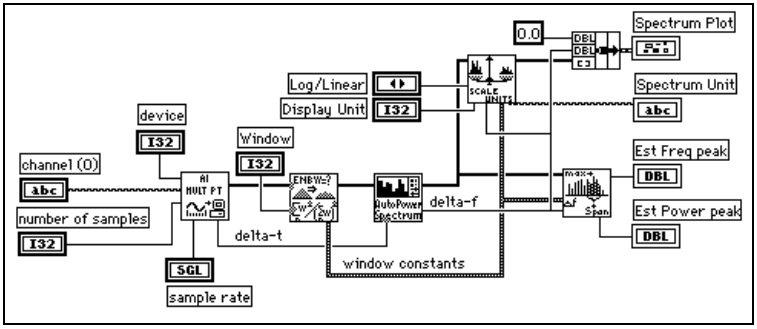
- Spectrum analysis applications
  - Amplitude and phase spectrum
  - Power spectrum
  - Scaled time domain window
  - Power and frequency estimate
  - Harmonic analysis and total harmonic distortion (THD) measurements
- Network and dual channel analysis applications
  - Impulse response function
  - Network functions (including coherence)
  - Cross power spectrum

The DFT, FFT, and power spectrum are useful for measuring the frequency content of stationary or transient signals. The FFT provides the average frequency content of the signal over the entire time that the signal was acquired. For this reason, you use the FFT mostly for stationary signal analysis (when the signal is not significantly changing in frequency content over the time that the signal is acquired), or when you want only the average energy at each frequency line. A large class of measurement problems fall in this category. For measuring frequency information that changes during the acquisition, you should use joint time-frequency analysis VIs, such as the Gabor Spectrogram.

The measurement VIs are built on top of the signal processing VIs and have the following characteristics, which model the behavior of traditional, benchtop frequency analysis instruments.

- Real-world, time-domain signal input is assumed.
- Outputs are in magnitude and phase, scaled, and in units where appropriate, ready for immediate graphing.
- Single-sided spectrums from DC to $\frac{\text{Sampling Frequency}}{2}$.
- Sampling period to frequency interval conversion for graphing with appropriate X axis units (in Hz).
- Corrections for the windows being used are applied where appropriate.
- Windows are scaled so that each window gives the same peak spectrum amplitude result within its amplitude accuracy constraints.
- Views power or amplitude spectrums in various unit formats, including decibels and spectral density units, such as V^2/Hz, $V/\sqrt{Hz}$, and so on.

In general, you can directly connect the measurement VIs to the output of data acquisition VIs and to graphs through the axis cluster, as the following spectrum analyzer diagram shows.



The measurement examples include the following:

- Amplitude Spectrum Example
- Simulated Dynamic Signal Analysis Example
- Total Harmonic Distortion (THD) Example

You can use the following examples with National Instruments hardware.

- Simple Spectrum Analyzer and Spectrum Analyzer—Both work with any analog input hardware (use dynamic signal acquisition hardware for good quality measurements).
- Dynamic Signal Analyzer and Network Analyzer—Both work with dynamic signal acquisition (DSA) hardware.

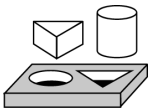
You Will Learn

- About the Measurement VIs and how they can be used to perform various signal processing operations.
- About how to calculate the frequency (amplitude & phase) spectrum of a time domain signal, with the appropriate units.
- About how to calculate the frequency response of a system, by processing the system stimulus and response signals, with the appropriate units.
- About how to compute the coherence function and how to use it to understand your frequency response measurements.
- About how to determine the total harmonic distortion present in a signal.

Spectrum Analysis

Calculating the Amplitude and Phase Spectrum of a Signal

In many applications, knowing the frequency content of a signal provides insight into the system that generated the signal. You can use the information obtained analyze the frequency content of sounds, calibrate instruments, estimate the amount of noise and vibration generated by parts of machines, and so on. The next activity demonstrates how to use the Amplitude and Phase Spectrum VI to measure the amplitude and phase of a signal.

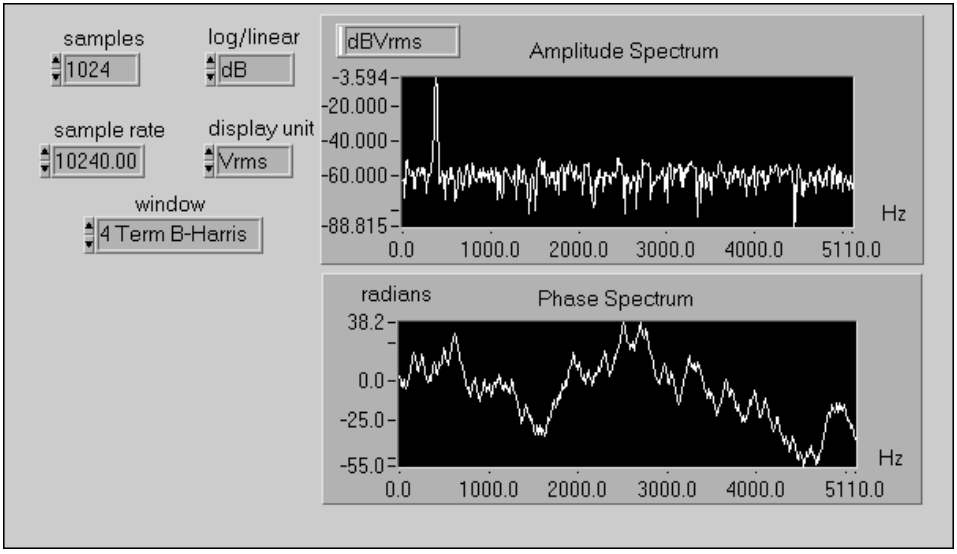


Activity 15-1. Use the Amplitude and Phase Spectrum VI

In this activity, your objective is to compute the amplitude and phase spectrum of a signal.

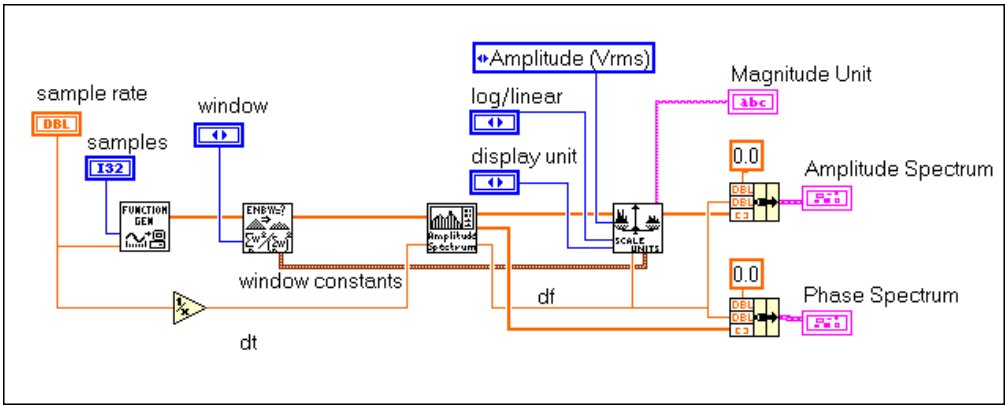
Front Panel

1. Open the Amp Spectrum Example VI found in the `examples\analysis\measure\measxmpl.llb` library. The signal is generated by the Simple Function Generator VI, which simulates a multi-function generator with additive white noise.



Block Diagram

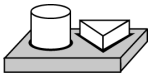
2. Open and examine the block diagram.



The Amplitude and Phase Spectrum VI calculates the amplitude spectrum and the phase spectrum of a time domain signal. The connections to this VI are discussed below.

The input time domain signal is applied at the Signal (V) control. The magnitude and phase of the spectrum of the input signal is available at the Amp Spectrum Mag (Vrms) and the Amp Spectrum Phase (radians) outputs respectively. The Spectrum Unit Conversion VI is used to convert the original Vrms output of the Amplitude and Phase Spectrum to any other common units (Vrms, Vpk, Vrms^2, Vpk^2, Vrms $\sqrt{Hz}$, Vpk $\sqrt{Hz}$, Vrms^2/Hz, and Vpk^2/Hz). The last four units are amplitude spectral density (Vrms $\sqrt{Hz}$, Vpk $\sqrt{Hz}$) and power spectral density (Vrms^2/Hz, and Vpk^2/Hz). The **window constants** output cluster from the Scaled Time Domain Window VI contains constants for the selected window which are required for spectral density measurements.

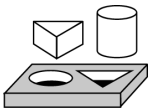
3. Run the VI.
4. Run the Amp Spectrum example continuously with the Simple Function Generator front panel open so that you can change the simulated frequency and waveform type as well as the amplitude and noise level of the signal. Notice the changes to the amplitude spectrum.



End of Activity 15-1.

Calculating the Frequency Response of a System

Measuring the frequency content of individual signals is useful on its own, but the frequency response of systems is widely used in analyzing the behavior of all kinds of networks, from the impedance of electrical components to the analysis of the natural vibrating frequency of dynamic structures. The frequency response completely characterizes how a network will respond to a given input.

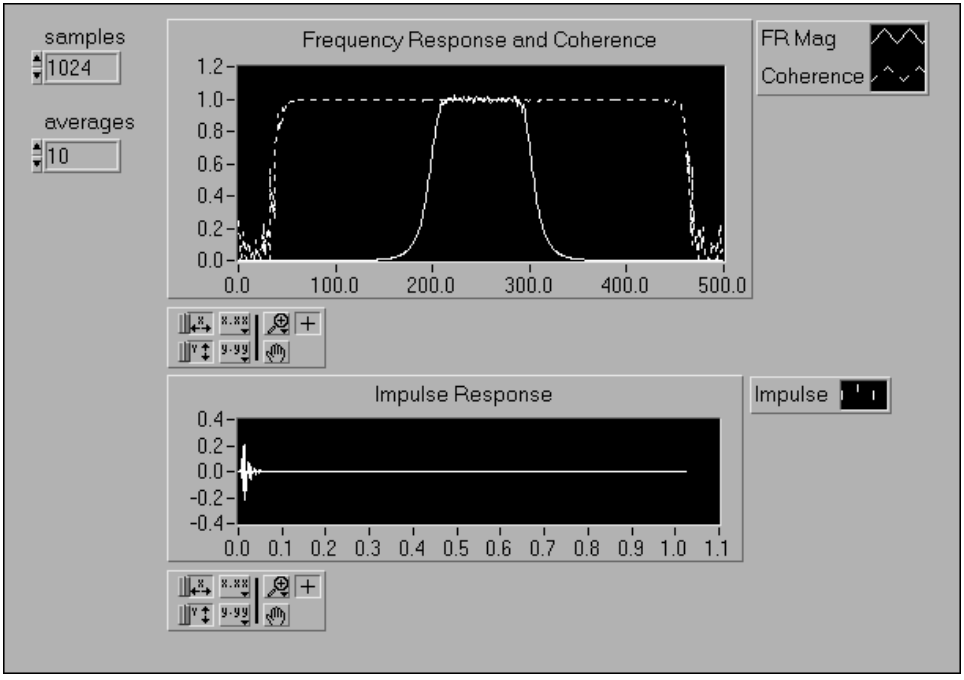


Activity 15-2. Compute the Frequency and Impulse Response

Your objective is to compute the frequency response and impulse response of a system, and to compute the coherence function and understand how it is used to validate your frequency response measurements.

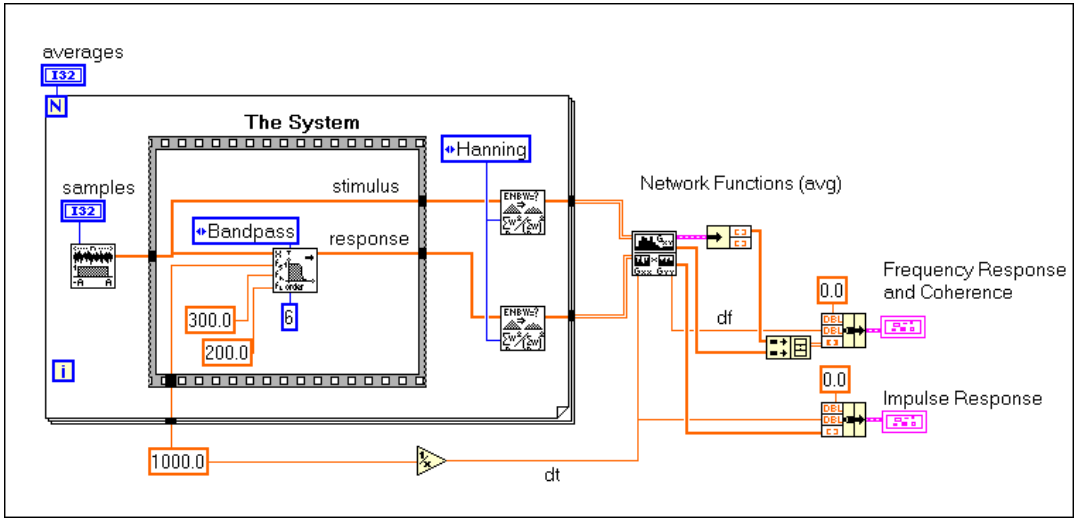
Front Panel

1. Open a new front panel and add the objects shown in the following illustration. This front panel shows the frequency response magnitude and the impulse response function for a bandpass filter. The coherence function is plotted on the same scale as the frequency response magnitude because it is also a spectral measurement.



Block Diagram

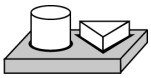
2. Open the block diagram and modify it as shown in the following illustration. Here we are measuring the system response of a bandpass filter (Butterworth Filter VI) by passing white noise (Uniform White Noise VI) as the system stimulus and collecting the filter output as the system response. Both the stimulus and response are windowed by the Hanning window (Scaled Time Domain Window VI) and the entire system is monitored for a number of frames, or *averages*. The stimulus and response data is then sent to the Network Functions (avg) VI where all the actual computations related to the system frequency response are made.



The Network Functions (avg) VI computes the frequency response (magnitude and phase), cross power spectrum (magnitude and phase), coherence function, and impulse response. By increasing the number of frames of input and output data (increasing averages on the front panel), the estimates of the system response functions improve. In this diagram, only the frequency response magnitude, coherence, and impulse response are plotted.

The coherence function measures how much of the output signal is correlated with the input signal, and thus it gives an indication of the validity of your frequency response estimate. Injected noise and nonlinear system behavior at certain frequencies cause the coherence function to dip below unity at those frequencies. For uncorrelated system noise, the more averages are taken, the more the coherence function approaches unity, and

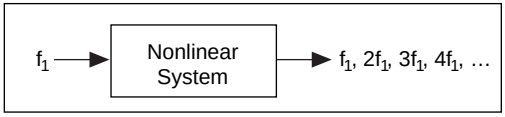
the better the frequency response estimate. One last bit of information to remember about the coherence function is that it is only defined when you are averaging more than one frame of input and output data. For only one average, coherence will be unity at all frequencies, even where your frequency response estimates may be poor.



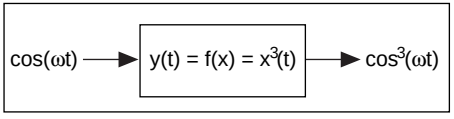
End of Activity 15-2.

Harmonic Distortion

When a signal, *x(t)*, of a particular frequency (for example, f1) is passed through a nonlinear system, the output of the system consists of not only the input frequency (f1), but also its harmonics (f2 = 2*f1, f3 = 3*f1, f4 = 4*f1, and so on). The number of harmonics, and their corresponding amplitudes, that are generated depends on the degree of nonlinearity of the system. In general, the more the nonlinearity, the higher the harmonics, and vice versa.



An example of a nonlinear system is a system where the output *y(t)* is the cube of the input signal *x(t)*.



So, if the input is

$$x(t) = \cos(\omega t),$$

the output is

$$x^3(t) = 0.5*\cos(\omega t) + 0.25*[\ \cos(\omega t) + \cos(3\omega t)\ ]$$

Therefore, the output contains not only the input fundamental frequency of ω, but also the third harmonic of 3ω.

Total Harmonic Distortion

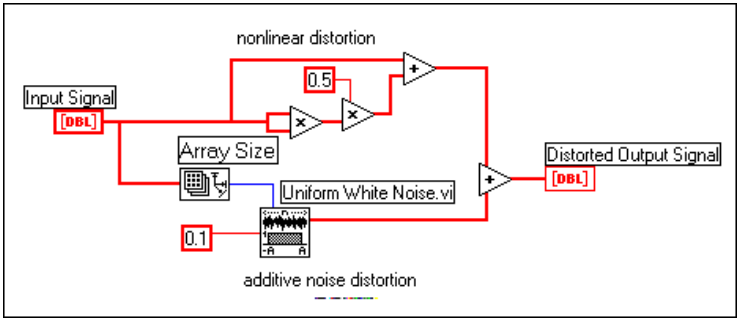
To determine the amount of nonlinear distortion that a system introduces, you need to measure the amplitudes of the harmonics that were introduced by the system relative to the amplitude of the fundamental. Harmonic distortion is a relative measure of the amplitudes of the harmonics as compared to the amplitude of the fundamental. If the amplitude of the fundamental is A_1, and the amplitudes of the harmonics are A_2 (second harmonic), A_3 (third harmonic), A_4 (fourth harmonic), ...A_N (*Nth* harmonic), then the total harmonic distortion (THD) is given by

$$THD = sqrt (A_1^2 + A_2^2 + A_3^2 + ... A_N^2)/A_1$$

and the percentage total harmonic distortion (% THD) is

$$\% THD = 100*sqrt (A_1^2 + A_2^2 + A_3^2 + ... A_N^2)/A_1$$

In the next activity, you will generate a sine wave and pass it through a nonlinear system. The block diagram of the nonlinear system is shown below:



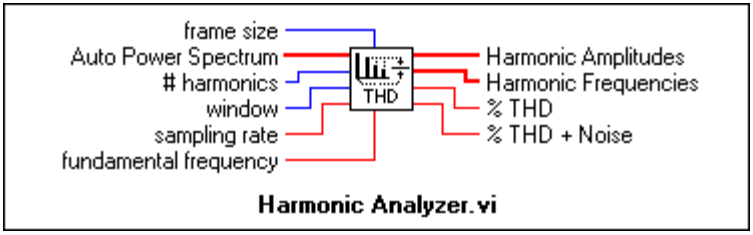
Verify from the block diagram that if the input is x(t) = cos(ωt), the output is

$$\begin{aligned} y(t) &= \cos(\omega t) + 0.5\cos^2(\omega t) + 0.1n(t) \\ &= \cos(\omega t) + [1 + \cos(2\omega t)]/4 + 0.1n(t) \\ &= 0.25 + \cos(\omega t) + 0.25\cos(2\omega t) + 0.1n(t) \end{aligned}$$

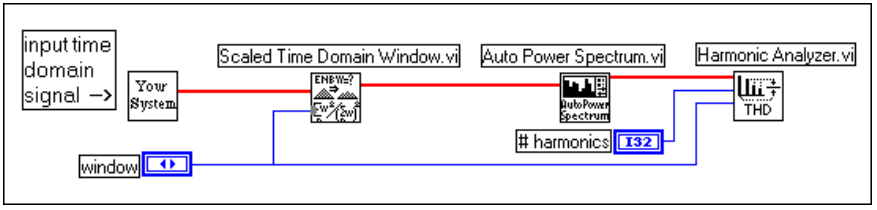
Therefore, this nonlinear system generates an additional DC component as well as the second harmonic of the fundamental.

Using the Harmonic Analyzer VI

You can use the Harmonic Analyzer VI to calculate the %THD present in the signal at the output of the nonlinear system. It finds the fundamental and harmonic components (their amplitudes and corresponding frequencies) present in the power spectrum applied at its input, and calculates the percentage of total harmonic distortion (%THD) and the percentage of total harmonic distortion plus noise (%THD + Noise). The connections to the Harmonic Analyzer VI are shown below:



To use this VI, you need to give it the power spectrum of the signal whose THD you want it to calculate. Thus, in this example, you need to make the following connections:



The Scaled Time Domain Window VI applies a window to the output y(t) of the nonlinear system (**Your System**). This is then passed on to the Auto Power Spectrum VI, which sends the power spectrum of y(t) to the Harmonic Analyzer VI, which then calculates the amplitudes and frequencies of the harmonics, the THD, and the %THD.

You can specify the number of harmonics you want the VI to find in the **# harmonics** control. Their amplitudes and corresponding frequencies are returned in the **Harmonic Amplitudes** and **Harmonic Frequencies** array indicators.

Note ***The number specified in the # harmonics control includes the fundamental. So, if you enter a value of 2 in the # harmonics control, it means to find the fundamental (say, of freq f1) and the second harmonic (of frequency f2 = 2*f1). If you enter a value of N, the VI will find the fundamental and the corresponding (N−1) harmonics.***

The following are explanations of some of the other controls:

fundamental frequency is an estimate of the frequency of the fundamental component. If left as zero (the default), the VI uses the frequency of the non-DC component with the highest amplitude as the fundamental frequency.

window is the type of window you applied to your original time signal. It is the window that you select in the Scaled Time Domain Window VI. For an accurate estimation of the THD, it is recommended that you select a window function. The default is the uniform window.

sampling rate is the input sampling frequency in Hz.

The *% THD + Noise* output requires some further explanation. The calculations for *% THD + Noise* are almost similar to that for *% THD*, except that the noise power is also added to that of the harmonics. It is given by

$$\% \text{ THD + Noise} = 100*\text{sqrt}(\text{sum(APS)})/A1$$

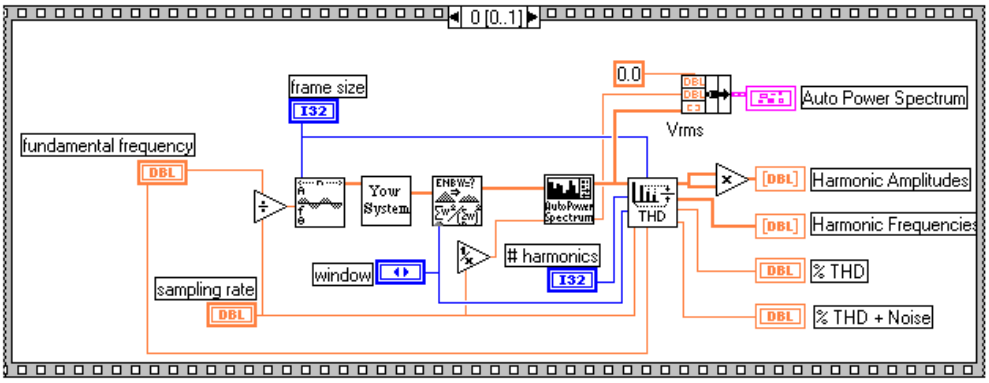
where sum(APS) is the sum of the Auto Power Spectrum elements minus the elements near DC and near the index of the fundamental frequency.

Activity 15-3. Calculate Harmonic Distortion

Your objective is to use the Harmonic Analyzer VI for harmonic distortion calculations.

Block Diagram

1. Open the THD Example VI from `examples\analysis\measxmpl.llb` and view the block diagram.

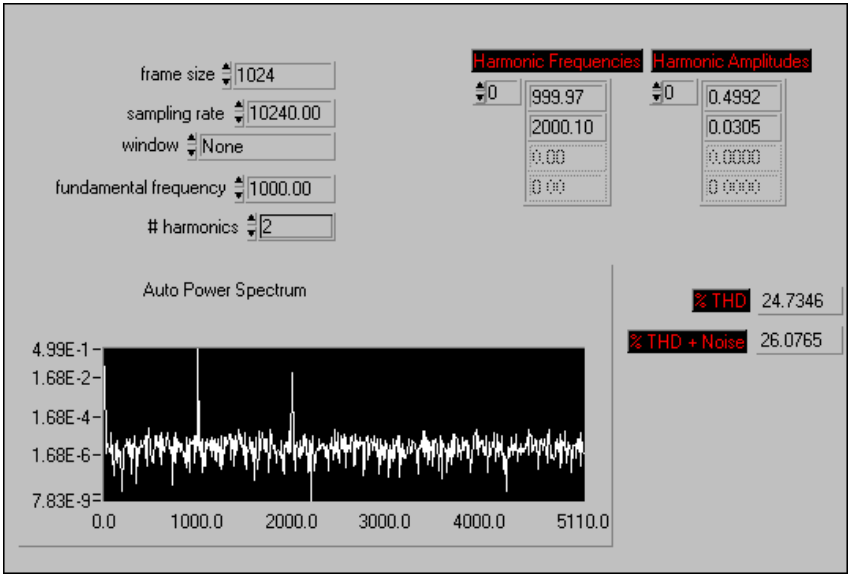


Some of this will already be familiar to you. Your System is the nonlinear system that you saw previously. Its output is windowed, and the power spectrum calculated and given to the Harmonic Analyzer VI.

The Sine Wave VI generates a fundamental of frequency specified in the **fundamental frequency** control.

Front Panel

2. View the front panel. At the bottom, you see a plot of the power spectrum of the output of the nonlinear system. On the top right side are the array indicators for the frequencies and amplitudes of the fundamental and its harmonics. The size of the array depends on the value entered in the **# harmonics** control.



3. Change the **fundamental frequency** to 1000, **# harmonics** to 2, and run the VI several times. Each time, note the values in the output indicators (**Harmonic Frequencies**, **Harmonic Amplitudes**, **% THD**, and **% THD + Noise**).

   Why do you get different values each time you run the VI?

   Which of the values, % THD or % THD + Noise, is larger? Can you explain why?

4. Run the VI with different selections of the **window** control and observe the peaks in the power spectrum.

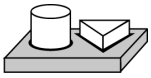
   Which window gives the narrowest peaks? The widest? Can you explain why?

5. Change the fundamental frequency to 3000 and run the VI.

   Why do you get an error?

   Hint: Consider the relationship between the Nyquist frequency and the frequency of the harmonic(s).

6. When you finish, close the VI and exit LabVIEW.



End of Activity 15-3.

Summary

You have seen that the measurement VIs can perform common measurement tasks. Some of these tasks include calculating the amplitude and phase spectrum of a signal, and the amount of harmonic distortion. Other VIs calculate properties of a system such as its transfer function, its impulse response, the cross power spectrum between the input and output signals, and so on. The ready-made VIs to perform these measurements are available in the **Analysis»Measurements** subpalette.

16

Filtering

This chapter explains how to filter unwanted frequencies from signals using infinite impulse response filters (IIR), finite impulse response filters (FIR), and nonlinear filters. For examples of how to use the analysis filter VIs, see the examples located in `examples\analysis\fltrxmpl.llb`.

Introduction to Digital Filtering Functions

Analog filter design is one of the most important areas of electronic design. Although analog filter design books featuring simple, tested filter designs exist, filter design is often reserved for specialists because it requires advanced mathematical knowledge and understanding of the processes involved in the system affecting the filter.

Modern sampling and digital signal processing tools make it possible to replace analog filters with digital filters in applications that require flexibility and programmability. These applications include audio, telecommunications, geophysics, and medical monitoring.

Digital filters have the following advantages over their analog counterparts:

- They are software programmable.
- They are stable and predictable.
- They do not drift with temperature or humidity and do not require precision components.
- They have a superior performance-to-cost ratio.

You can use digital filters in LabVIEW to control parameters such as filter order, cutoff frequencies, amount of ripple, and stopband attenuation.

The digital filter VIs described in this section follow the virtual instrument philosophy. The VIs handle all the design issues, computations, memory management, and actual data filtering internally, transparent to the user. You do not have to be an expert in digital filters or digital filter theory to process the data.

The following discussion of sampling theory is intended to give you a better understanding of the filter parameters and how they relate to the input parameters.

The sampling theorem states that you can reconstruct a continuous-time signal from discrete, equally spaced samples if the sampling frequency is at least twice that of the highest frequency in the time signal. Assume you can sample the time signal of interest at Δt equally spaced intervals without losing information. The Δt parameter is the sampling interval.

You can obtain the sampling rate or sampling frequency f_s from the sampling interval

$$f_s = \frac{1}{\Delta t},$$

which means that, according to the sampling theorem, the highest frequency that the digital system can process is

$$f_{Nyq} = \frac{f_s}{2}.$$

The highest frequency the system can process is known as the Nyquist frequency. This also applies to digital filters. For example, if your sampling interval is

$$\Delta t = 0.001 \text{ sec},$$

then the sampling frequency is

$$f_s = 1{,}000 \text{ Hz},$$

and the highest frequency that the system can process is

$$f_{Nyq} = 500 \text{ Hz}.$$

The following types of filtering operations are based upon filter design techniques:

- Smoothing windows
- Infinite impulse response (IIR) or recursive digital filters
- Finite impulse response (FIR) or nonrecursive digital filters
- Nonlinear filters

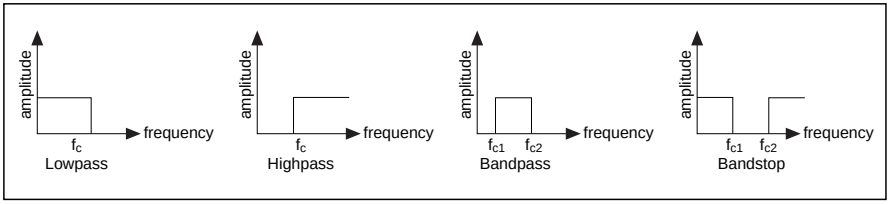
The rest of this chapter presents a brief theoretical background on the IIR, FIR, and nonlinear techniques and discusses the digital filter VIs corresponding to each technique. Refer to Chapter 14, *Smoothing Windows*, for information about the VIs that implement smoothing windows.

Ideal Filters

Filters alter or remove unwanted frequencies. Depending on the frequency range that they either pass or attenuate, they can be classified into the following types:

- *A lowpass filter* passes low frequencies, but attenuates high frequencies.
- A *highpass filter* passes high frequencies, but attenuates low frequencies.
- A *bandpass filter* passes a certain band of frequencies.
- A *bandstop filter* attenuates a certain band of frequencies.

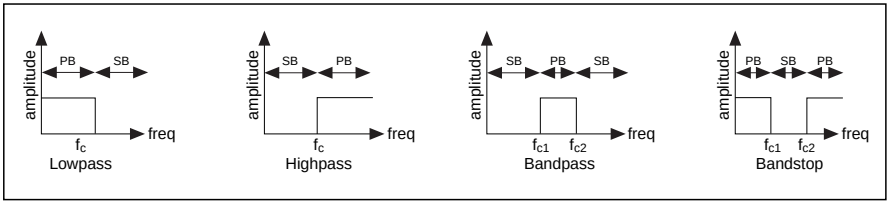
The ideal frequency response of these filters is shown below:



You see that the lowpass filter passes all frequencies below f_c, whereas the highpass filter passes all frequencies above f_c. The bandpass filter passes all frequencies between f_{c1} and f_{c2}, whereas the bandstop filter attenuates all frequencies between f_{c1} and f_{c2}. The frequency points f_c, f_{c1} and f_{c2} are known as the cut-off frequencies of the filter. When designing filters, you need to specify these cut-off frequencies.

The frequency range that is passed through the filter is known as the *passband* (PB) of the filter. An ideal filter has a gain of one (0 dB) in the passband so that the amplitude of the signal neither increases nor decreases. The *stopband* (SB) corresponds to that range of frequencies that do not pass

through the filter at all and are rejected (attenuated). The passband and the stopband for the different types of filters are shown below:

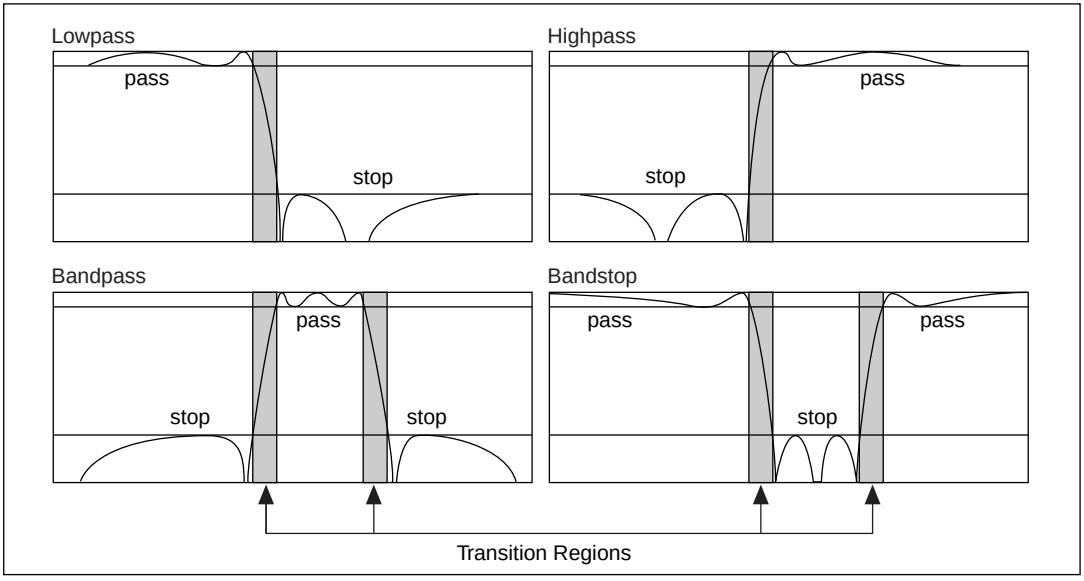


Note that whereas the lowpass and highpass filters have one passband and one stopband, the bandpass filter has one passband and two stopbands, and the bandstop filter has two passbands and one stopband.

Practical (Nonideal) Filters

The Transition Band

Ideally, a filter should have a unit gain (0 dB) in the passband, and a gain of zero (–infinity dB) in the stopband. However, in a real implementation, not all of these criteria can be fulfilled. In practice, there is always a finite transition region between the passband and the stopband. In this region, the gain of the filter changes gradually from 1 (0 dB) in the passband to 0 (–infinity) in the stopband. The following diagrams show the passband, the stopband, and the transition region (TR) for the different types of nonideal filters. Note that the passband is now the region where the frequency range within which the gain of the filter varies from 0 dB to –3 dB.



Passband Ripple and Stopband Attenuation

In many applications, it is okay to allow the gain in the passband to vary slightly from unity. This variation in the passband is called the *passband ripple* and is the difference between the actual gain and the desired gain of unity. The *stopband attenuation*, in practice, cannot be infinite, and you must specify a value with which you are satisfied. Both the passband ripple and the stopband attenuation are measured in decibels or dB, defined by:

$$dB = 20*\log_{10}(A_o(f)/A_i(f))$$

where $\log_{10}$ denotes the logarithm to the base 10, and $A_i(f)$ and $A_o(f)$ are the amplitudes of a particular frequency *f* before and after the filtering, respectively.

For example, for –0.02 dB passband ripple, the formula gives:

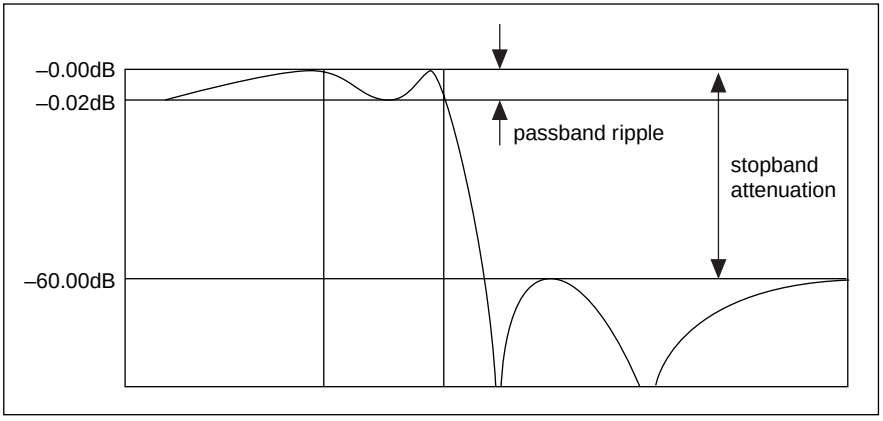
$$-0.02 = 20*\log_{10}(A_o(f)/A_i(f))$$
$$A_o(f)/A_i(f) = 10^{-0.001} = 0.9977$$

which shows that the ratio of input and output amplitudes is close to unity.

If you have –60 dB attenuation in the stopband, you have

$$-60 = 20*\log_{10}(A_o(f)/A_i(f))$$
$$A_o(f)/A_i(f) = 10^{-3} = 0.001$$

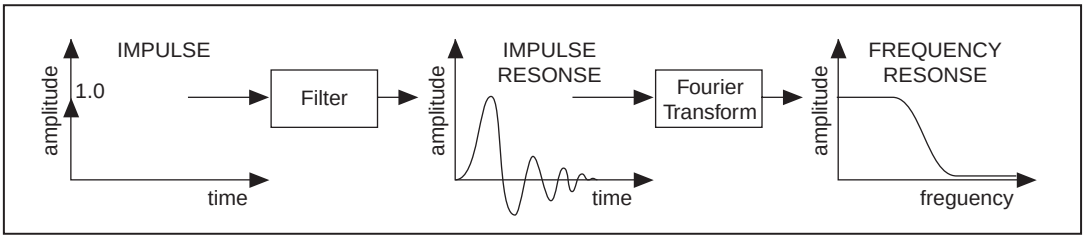
which means the output amplitude is 1/1000 of the input amplitude. The following figure, though not drawn to scale, illustrates this concept



Note ***Attenuation is usually expressed in decibels without the word "minus," but a negative dB value is normally assumed.***

IIR and FIR Filters

Another method of classification of filters is based on their impulse response. But what is an impulse response? The response of a filter to an input that is an impulse ($x[0] = 1$ and $x[i] = 0$ for all $i \neq 0$) is called the *impulse response* of the filter (see figure below). The Fourier transform of the impulse response is known as the *frequency response* of the filter. The frequency response of a filter tells you what the output of the filter is going to be at different frequencies. In other words, it tells you the gain of the filter at different frequencies. For an ideal filter, the gain should be 1 in the passband and 0 in the stopband. So, all frequencies in the passband are passed "as is" to the output, but there is no output for frequencies in the stopband.



If the impulse response of the filter falls to zero after a finite amount of time, it is known as a *finite impulse response (FIR)* filter. However, if the impulse response exists indefinitely, it is known as an *infinite impulse response (IIR)* filter. Whether the impulse response is finite or not (that is, whether the filter is FIR or IIR) depends on how the output is calculated.

The basic difference between FIR and IIR filters is that for FIR filters, the output depends only on the current and past input values, whereas for IIR filters, the output depends not only on the current and past input values, but also on the past output values.

As an example, consider a cash register at a supermarket. Let $x[k]$ be the cost of the present item that a customer buys and x(k–1) is the price of the previous item, where $1 \leq k \leq N$, and N is the total number of items. The cash register adds the cost of each item to produce a "running" total. This "running" total y[k], up to the k^{th} item, is given by

$$y[k] = x[k] + x[k–1] + x[k–2] + x[k–3] + ...+ x[1] \qquad (16\text{-}1A)$$

Thus, the total for N items is *y[N]*. Because *y[k]* is the total up to the k^{th} item, and *y[k–1]* is the total up to the *(k–1)*st item, you can rewrite Equation 16-1A as

$$y[k] = y[k–1] + x[k] \qquad (16\text{-}1B)$$

If you add a sales tax of 8.25%, Equations 16-1A and 16-1B can be rewritten as

$$y[k] = 1.0825x[k] + 1.0825x[k–1] + 1.0825\,x[k–2] + 1.0825x[k–3] + ... + 1.0825x[1] \qquad (16\text{-}2A)$$

$$y[k] = y[k–1] + 1.0825x[k] \qquad (16\text{-}2B)$$

Note that both Equations 16-2A and 16-2B are identical in describing the behavior of the cash register. The difference is that whereas 16-2A is implemented only in terms of the inputs, 16-2B is implemented in terms of both the input and the output. Equation 16-2A is known as the *nonrecursive*, or FIR, implementation. Equation 16-2B is known as the *recursive*, or IIR, implementation.

Filter Coefficients

In Equation 16-2A, the multiplying constant for each term is 1.0825. In Equation 16-2B, the multiplying constants are 1 (for *y[k–1]*) and 1.0825 (for *x[k]*). These multiplying constants are known as the *coefficients* of the filter. For an IIR filter, the coefficients multiplying the inputs are known as the *forward coefficients*, and those multiplying the outputs are known as the *reverse coefficients*.

Equations of the form 16-1A, 16-1B, 16-2A, or 16-2B that describe the operation of the filter are known as *difference* equations.

The disadvantage of IIR filters is that the phase response is nonlinear. If the application does not require phase information, such as simple signal monitoring, IIR filters may be appropriate. You should use FIR filters for those applications requiring linear phase responses. The recursive nature of IIR filters makes them more difficult to design and implement.

Infinite Impulse Response Filters

Infinite impulse response filters (IIR) are digital filters with impulse responses that can theoretically be infinite in length (duration). The general difference equation characterizing IIR filters is

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{N_b-1} b_j x_{i-j} - \sum_{k=1}^{N_a-1} a_k y_{i-k}\right) \qquad (16\text{-}3)$$

where N_b is the number of *forward* coefficients (b_j) and N_a is the number of *reverse* coefficients (a_k).

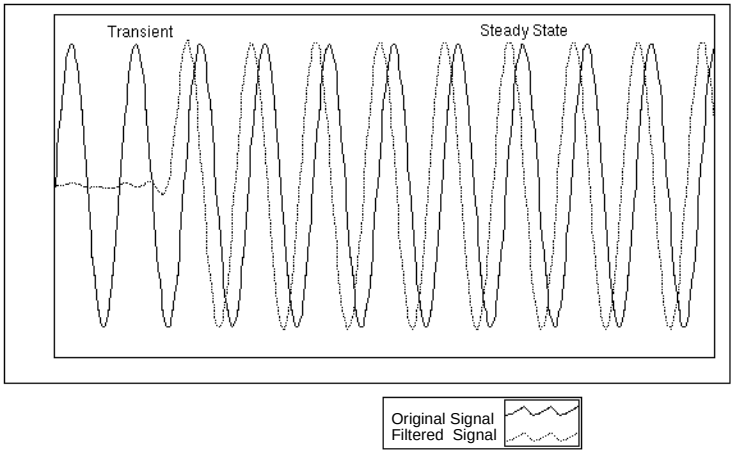
In most IIR filter designs (and in all of the LabVIEW IIR filters), coefficient a_0 is 1. The output sample at the present sample index *i* is the sum of scaled present and past inputs (x_i and x_{i-j} when $\neq 0$) and scaled past outputs (y_{i-k}). Because of this, IIR filters are also known as recursive filters or autoregressive moving-average (ARMA) filters.

The response of the general IIR filter to an impulse ($x_0 = 1$ and $x_i = 0$ for all $i \neq 0$) is called the impulse response of the filter. The impulse response of the filter described by Equation 16-3 is indeed of infinite length for nonzero coefficients. In practical filter applications, however, the impulse response of stable IIR filters decays to near zero in a finite number of samples.

IIR filters in LabVIEW contain the following properties:

- Negative indices resulting from Equation 16-3 are assumed to be zero the first time you call the VI.
- Because the initial filter state is assumed to be zero (negative indices), a transient proportional to the filter order occurs before the filter reaches a steady state. The duration of the transient response, or delay, for lowpass and highpass filters is equal to the filter order.
- Delay = order.
- The duration of the transient response for bandpass and bandstop filters is twice the filter order
- Delay = 2 * order.

You can eliminate this transient response on successive calls by enabling state memory. To enable state memory, set the **init/cont** control of the VI to TRUE (continuous filtering).



The number of elements in the filtered sequence equals the number of elements in the input sequence.

The filter retains the internal filter state values when the filtering completes.

The advantage of digital IIR filters over finite impulse response (FIR) filters is that IIR filters usually require fewer coefficients to perform similar filtering operations. Thus, IIR filters execute much faster and do not require extra memory, because they execute in place.

The disadvantage of IIR filters is that the phase response is nonlinear. If the application does not require phase information, such as simple signal monitoring, IIR filters may be appropriate. You should use FIR filters for those applications requiring linear phase responses.

Cascade Form IIR Filtering

Filters implemented using the structure defined by Equation 16-4 directly are known as *direct form* IIR filters. Direct form implementations are often sensitive to errors introduced by coefficient quantization and by computational, precision limits. Additionally, a filter designed to be stable can become unstable with increasing coefficient length, which is proportional to filter order.

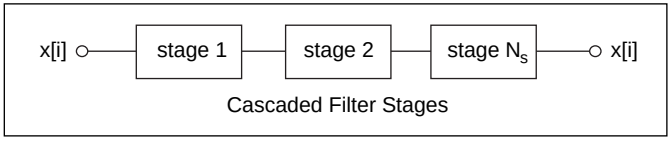
A less sensitive structure can be obtained by breaking up the direct form transfer function into lower order sections, or filter stages. The direct form transfer function of the filter given by Equation 16-4 (with $a_0 = 1$) can be written as a ratio of z transforms, as follows:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_{N_b - 1} z^{-(N_b - 1)}}{1 + a_1 z^{-1} + \ldots + a_{N_a - 1} z^{-(N_a - 1)}} \quad . \tag{16-4}$$

By factoring Equation 16-4 into second-order sections, the transfer function of the filter becomes a product of second-order filter functions

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}} \tag{16-5}$$

where $N_s = \lfloor N_a/2 \rfloor$ is the largest integer $\leq N_a/2$, and $N_a \geq Nb$. (N_s is the *number of stages*.) This new filter structure can be described as a *cascade* of second-order filters.



Each individual stage is implemented using the *direct form II* filter structure because it requires a minimum number of arithmetic operations and a minimum number of delay elements (internal filter states). Each stage has one input, one output, and two past internal states ($s_k[i–1]$ and $s_k[i–2]$).

If n is the number of samples in the input sequence, the filtering operation proceeds as in the following equations:

$$y_0[i] = x[i],$$

$$s_k[i] = y_{k-1}[i–1] – a_{1k}s_k[i–1] – a_{2k}s_k[i–2],\ k = 1, 2,..., N_s$$

$$y_k[i] = b_{Ok}s_k[i] + b_{1k}s_k[i–1] + b_{2k}s_k[i–2],\ k = 1, 2,..., N_s$$

$$y[i] = y_{Ns}[i]$$

for each sample

$$i = 0, 1, 2,...,n–1.$$

For filters with a single cutoff frequency (lowpass and highpass), second-order filter stages can be designed directly. The overall IIR lowpass or highpass filter contains cascaded second-order filters.

For filters with two cutoff frequencies (bandpass and bandstop), fourth-order filter stages are a more natural form. The overall IIR bandpass or bandstop filter is cascaded fourth-order filters. The filtering operation for fourth-order stages proceeds as in the following equations:

$$y_0[i] = x[i],$$

$$s_k[i] = y_{k-1}[i–1] – a_{1k}s_k[i–1] – a_{2k}s_k[i–2] – a_{3k}s_k[i–3] – a_{4k}s_k[i–4],\ k = 1, 2,..., N_s$$

$$y_k[i] = b_{0k}s_k[i] + b_{1k}s_k[i–1] + b_{2k}s_k[i–2] + b_{3k}s_k[i–3] + b_{4k}s_k[i–4],\ k = 1, 2,..., N_s$$

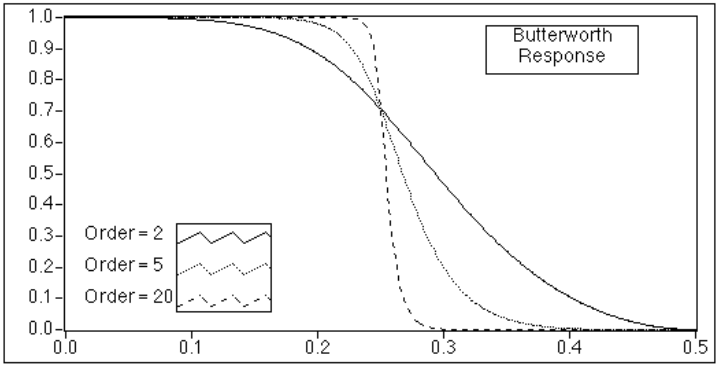
$$y[i] = y_{Ns}[i].$$

Notice that in the case of fourth-order filter stages, $N_s = \lfloor (N_a + 1)/4 \rfloor$.

Butterworth Filters

A smooth response at all frequencies and a monotonic decrease from the specified cutoff frequencies characterize the frequency response of Butterworth filters. Butterworth filters are maximally flat—the ideal response of unity in the passband and zero in the stopband. The half power frequency or the 3-dB down frequency corresponds to the specified cutoff frequencies.

The following illustration shows the response of a lowpass Butterworth filter. The advantage of Butterworth filters is a smooth, monotonically decreasing frequency response. After you set the cutoff frequency, LabVIEW sets the *steepness* of the transition proportional to the filter order. Higher order Butterworth filters approach the ideal lowpass filter response.

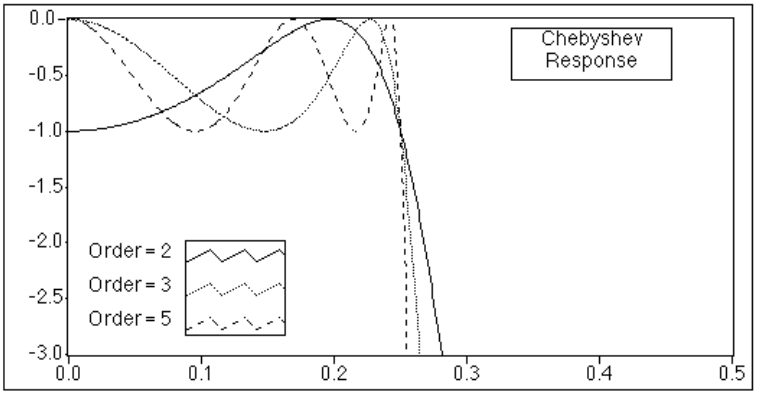


Chebyshev Filters

Butterworth filters do not always provide a good approximation of the ideal filter response because of the slow rolloff between the passband (the portion of interest in the spectrum) and the stopband (the unwanted portion of the spectrum).

Chebyshev filters minimize peak error in the passband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want (the maximum tolerable error in the passband). The frequency response characteristics of Chebyshev filters have an equiripple magnitude response in the passband, monotonically decreasing magnitude response in the stopband, and a sharper rolloff than Butterworth filters.

The following graph shows the response of a lowpass Chebyshev filter. Notice that the equiripple response in the passband is constrained by the maximum tolerable ripple error and that the sharp rolloff appears in the stopband. The advantage of Chebyshev filters over Butterworth filters is that Chebyshev filters have a sharper transition between the passband and the stopband with a lower order filter. This produces smaller absolute errors and higher execution speeds.



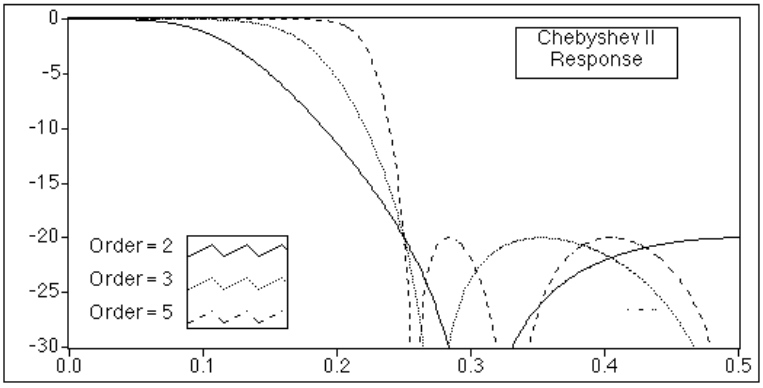
Chebyshev II or Inverse Chebyshev Filters

Chebyshev II, also known as inverse Chebyshev or Type II Chebyshev filters, are similar to Chebyshev filters, except that Chebyshev II filters distribute the error over the stopband (as opposed to the passband), and Chebyshev II filters are maximally flat in the passband (as opposed to the stopband).

Chebyshev II filters minimize peak error in the stopband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want. The frequency response characteristics of Chebyshev II filters are equiripple magnitude response in the stopband, monotonically decreasing magnitude response in the passband, and a rolloff sharper than Butterworth filters.

The following graph plots the response of a lowpass Chebyshev II filter. Notice that the equiripple response in the stopband is constrained by the maximum tolerable error and that the smooth monotonic rolloff appears in the stopband. The advantage of Chebyshev II filters over Butterworth filters is that Chebyshev II filters give a sharper transition between the passband and the stopband with a lower order filter. This difference corresponds to a smaller, absolute error and higher execution speed. One advantage of

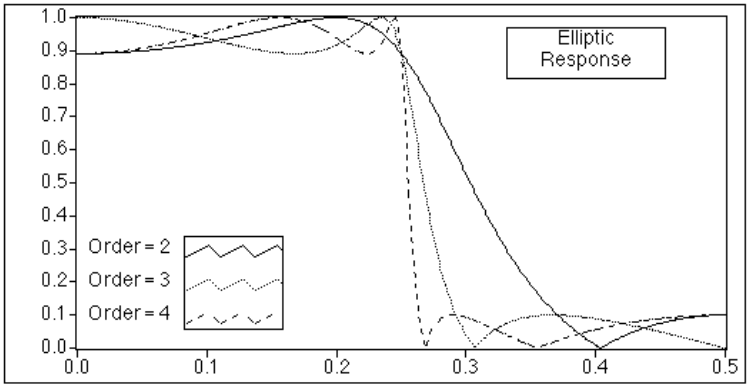
Chebyshev II filters over regular Chebyshev filters is that Chebyshev II filters distribute the error in the stopband instead of the passband.



Elliptic (or Cauer) Filters

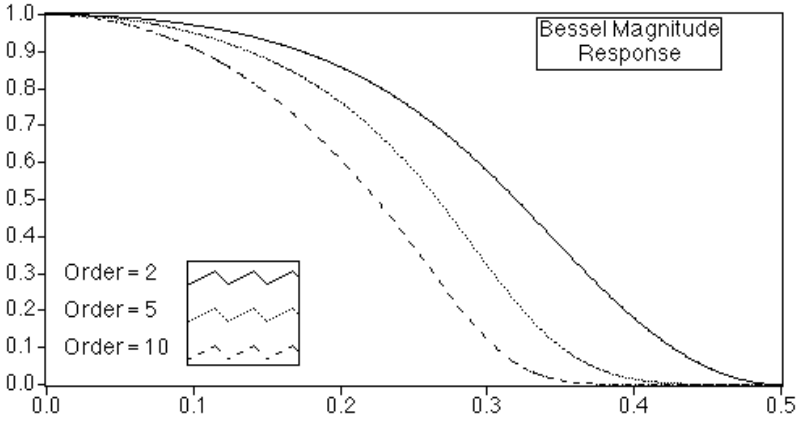
Elliptic filters minimize the peak error by distributing it over the passband and the stopband. Equi-ripples in the passband and the stopband characterize the magnitude response of elliptic filters. Compared with the same order Butterworth or Chebyshev filters, the elliptic design provides the sharpest transition between the passband and the stopband. For this reason, elliptic filters are used widely.

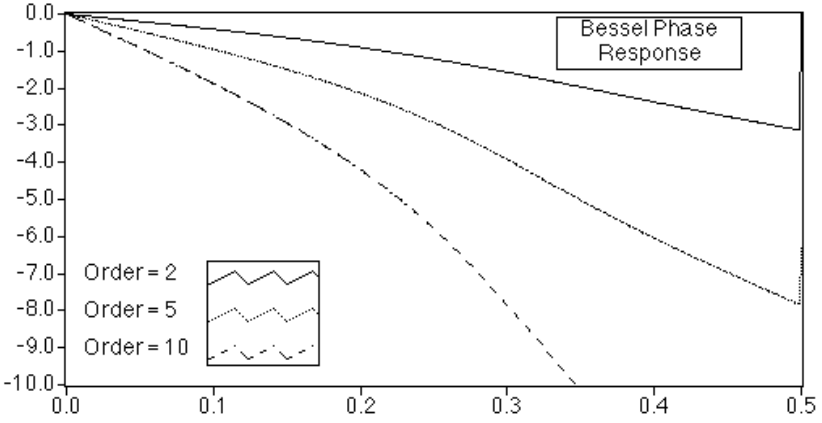
The following graph plots the response of a lowpass elliptic filter. Notice that the ripple in both the passband and stopband is constrained by the same maximum tolerable error (as specified by ripple amount in dB). Also, notice the sharp transition edge for even low-order elliptic filters.



Bessel Filters

You can use Bessel filters to reduce nonlinear phase distortion inherent in all IIR filters. In higher order filters and those with a steeper rolloff, this condition is more pronounced, especially in the transition regions of the filters. Bessel filters have maximally flat response in both magnitude and phase. Furthermore, the phase response in the passband of Bessel filters, which is the region of interest, is nearly linear. Like Butterworth filters, Bessel filters require high-order filters to minimize the error and, for this reason, are not widely used. You can also obtain linear phase response using FIR filter designs.The following graphs plot the response of a lowpass Bessel filter. Notice that the response is smooth at all frequencies, as well as monotonically decreasing in both magnitude and phase. Also, notice that the phase in the passband is nearly linear.





Finite Impulse Response Filters

Finite impulse response (FIR) filters are digital filters, which have a finite impulse response. FIR filters are also known as nonrecursive filters, convolution filters, or moving-average (MA) filters because you can express the output of an FIR filter as a finite convolution

$$y_i = \sum_{k=0}^{n-1} h_k x_{i-k}$$

where x represents the input sequence to be filtered, y represents the output filtered sequence, and h represents the FIR filter coefficients.

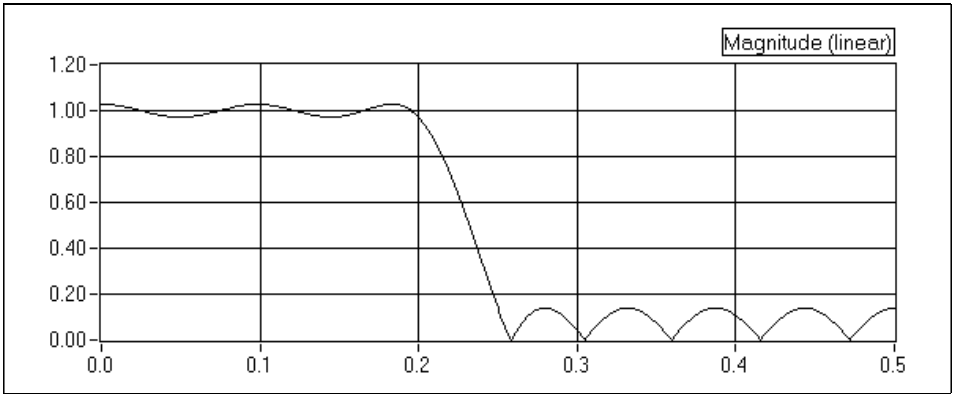
The following list gives the most important characteristics of FIR filters:

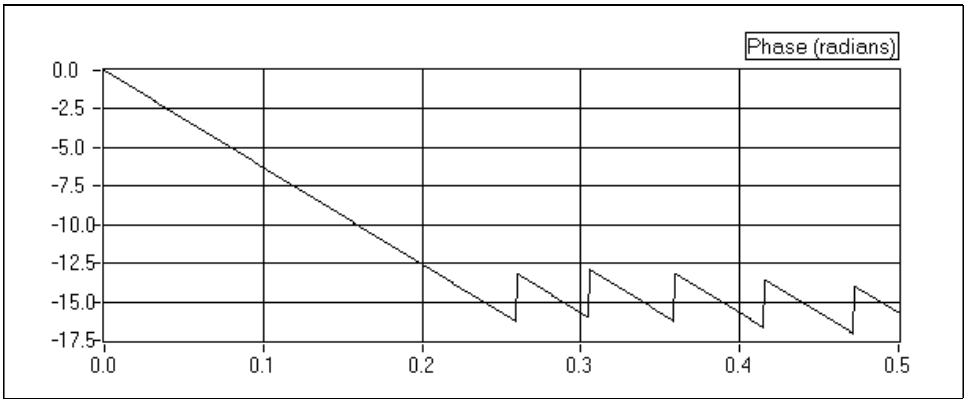
- They can achieve linear phase because of filter coefficient symmetry in the realization.
- They are always stable.
- You can perform the filtering function using the convolution and, as such, generally associate a delay with the output sequence

  $$\text{delay} = \frac{n-1}{2},$$

  where n is the number of FIR filter coefficients.

The following graphs plot a typical magnitude and phase response of FIR filters versus normalized frequency.





The discontinuities in the phase response arise from the discontinuities introduced when you compute the magnitude response using the absolute value. Notice that the discontinuities in phase are on the order of *pi*. The phase, however, is clearly linear. See Appendix A, *Analysis References*, for material that can give you more information on this topic.

You design FIR filters by approximating a specified, desired frequency response of a discrete-time system. The most common techniques approximate the desired magnitude response while maintaining a linear-phase response.

Designing FIR Filters by Windowing

The simplest method for designing linear-phase FIR filters is the *window design* method. To design a FIR filter by windowing, you start with an ideal frequency response, calculate its impulse response, and then truncate the impulse response to produce a finite number of coefficients. To meet the linear-phase constraint, by maintain symmetry about the center point of the coefficients. The truncation of the ideal impulse response results in the effect known as the Gibbs phenomenon—oscillatory behavior near abrupt transitions (cutoff frequencies) in the FIR filter frequency response.

You can reduce the effects of the Gibbs phenomenon by smoothing the truncation of the ideal impulse response using a smoothing window function. By tapering the FIR coefficients at each end, you can diminish the height of the side lobes in the frequency response. The disadvantage to this method, however, is that the main lobe widens, resulting in a wider transition region at the cutoff frequencies. The selection of a window function, then, is similar to the choice between Chebyshev and Butterworth

IIR filters in that it is a trade-off between side lobe levels near the cutoff frequencies and width of the transition region.

Designing FIR filters by windowing is simple and computationally inexpensive. It is therefore the fastest way to design FIR filters. It is not necessarily, however, the best FIR filter design method.

Designing Optimum FIR Filters Using the Parks-McClellan Algorithm

The Parks-McClellan algorithm offers an optimum FIR filter design technique that attempts to design the best filter possible for a given number of coefficients. Such a design reduces the adverse effects at the cutoff frequencies. It also offers more control over the approximation errors in different frequency bands—control that is not possible with the window method.

Using the Parks-McClellan algorithm to design FIR filters is computationally expensive. This method, however, produces optimum FIR filters by applying time-consuming iterative techniques.

Designing Narrowband FIR Filters

When you use conventional techniques to design FIR filters with especially narrow bandwidths, the resulting filter lengths may be very long. FIR filters with long filter lengths often require lengthy design and implementation times, and are more susceptible to numerical inaccuracy. In some cases, conventional filter design techniques, such as the Parks-McClellan algorithm, may fail the design altogether.

You can use a very efficient algorithm, called the Interpolated Finite Impulse Response (IFIR) filter design technique, to design narrowband FIR filters. Using this technique produces narrowband filters that require far fewer coefficients (and therefore fewer computations) than those filters designed by the direct application of the Parks-McClellan algorithm. LabVIEW also uses this technique to produce wideband, lowpass (cutoff frequency near Nyquist) and highpass filters (cutoff frequency near zero). For more information about IFIR filter design, see *Multirate Systems and Filter Banks* by P.P. Vaidyanathan, or the paper on interpolated finite impulse response filters by Neuvo, et al., listed in Appendix A, *Analysis References*, of this manual.

Windowed FIR Filters

You use the **filter type** parameter of the FIR VIs to select the type of windowed FIR filter you want: lowpass, highpass, bandpass, or bandstop. The following list gives the two related FIR VIs:

- FIR Windowed Coefficients—Generates the windowed (or unwindowed) coefficients.
- FIR Windowed Filters—Filters the input using windowed (or unwindowed) coefficients.

Optimum FIR Filters

You can use the Parks-McClellan algorithm to design optimum, linear-phase, FIR filter coefficients in the sense that the resulting filter optimally matches the filter specifications for a given number of coefficients. The Parks-McClellan VI takes as input an array of band descriptions, each containing information describing the response you want for the given band. The VI outputs the FIR coefficients along with computed ripple, which is a measure of the deviation of the resulting filter from the ideal filter specifications.

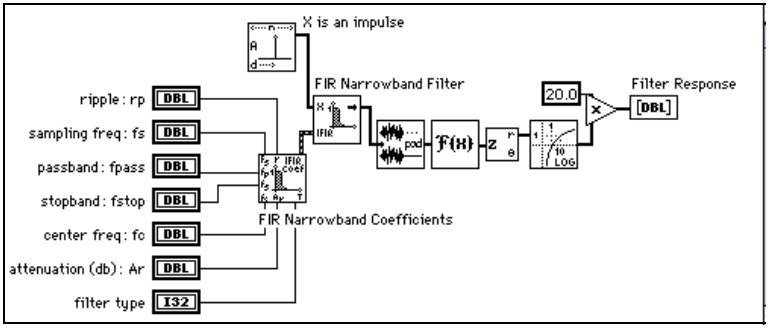
Four VIs use the Parks-McClellan VI to implement filters whose stopband and passband ripple level are equal: Equiripple LowPass, Equiripple HighPass, Equiripple BandPass, and Equiripple BandStop.

FIR Narrowband Filters

You can design narrowband FIR filters using the FIR Narrowband Coefficients VI, and then implement the filtering using the FIR Narrowband Filter VI. The design and implementation are separate operations because many narrowband filters require lengthy design times, while the actual filtering process is very fast and efficient. Keep this in mind when creating your narrowband filtering diagrams.

The parameters required for narrowband filter specification are filter type, sampling rate, passband and stopband frequencies, passband ripple (linear scale), and stopband attenuation (decibels). For bandpass and bandstop filters, passband and stopband frequencies refer to bandwidths, and you must specify an additional center frequency parameter. You can also design wideband lowpass filters (cutoff frequency near Nyquist) and wideband highpass filters (cutoff frequency near zero) using the narrowband filter VIs.

The following illustration shows how to use the FIR Narrowband Coefficients VI and the FIR Narrowband Filter VI to estimate the response of a narrowband filter to an impulse.



Nonlinear Filters

Smoothing windows, IIR filters, and FIR filters are linear because they satisfy the superposition and proportionality principles

$$L\{ax(t) + by(t)\} = aL\{x(t)\} + bL\{y(t)\},$$

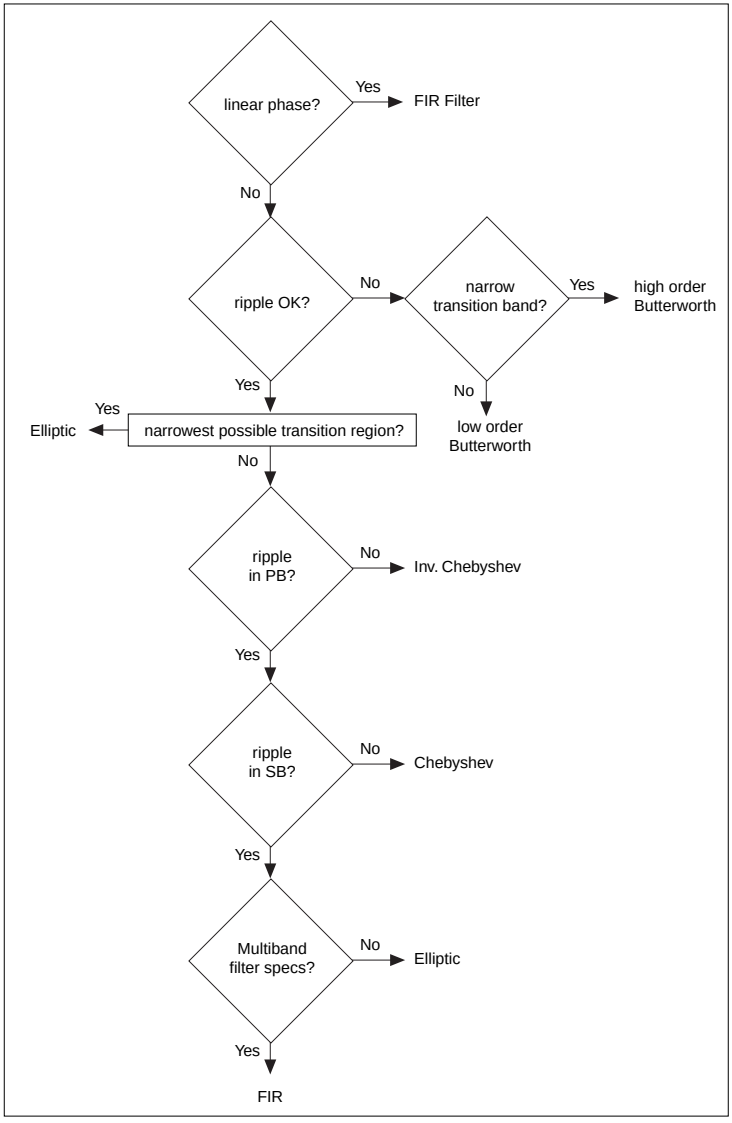
where a and b are constants, $x(t)$ and $y(t)$ are signals, $L\{\bullet\}$ is a linear filtering operation, and their inputs and outputs are related via the convolution operation.

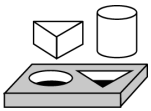
A nonlinear filter does not meet the preceding conditions and you cannot obtain its output signals via the convolution operation, because a set of coefficients cannot characterize the impulse response of the filter. Nonlinear filters provide specific filtering characteristics that are difficult to obtain using linear techniques. The median filter is a nonlinear filter that combines lowpass filter characteristics (to remove high-frequency noise) and high-frequency characteristics (to detect edges).

How Do I Decide Which Filter to Use?

Now that you have seen the different types of filters and their characteristics, the question arises as to which filter design is best suited for your application. In general, some of the factors affecting the choice of a suitable filter are whether you require linear phase, whether you can tolerate ripples, and whether a narrow transition band is required. The following flowchart is expected to serve as a guideline for selecting the

correct filter. Keep in mind that in practice, you may need to experiment with several different options before finally finding the best one.



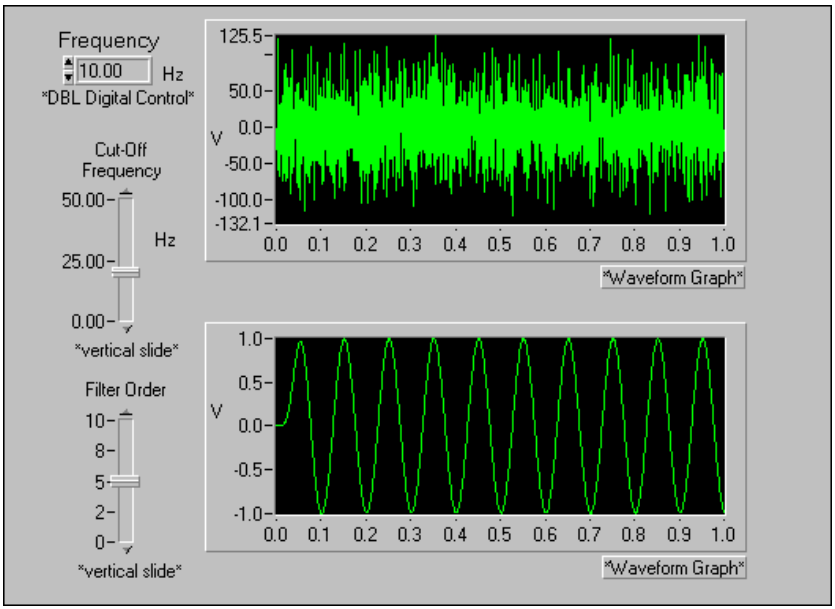


Activity 16-1. Extract a Sine Wave

Your objective is to filter data samples that consist of both high-frequency noise and a sinusoidal signal.

In this activity, you combine a sine wave generated by the Sine Pattern VI with high-frequency noise. (The high-frequency noise is obtained by highpass filtering uniform white noise with a Butterworth filter.) The combined signal is then lowpass filtered by another Butterworth filter to extract the sine wave.

Front Panel

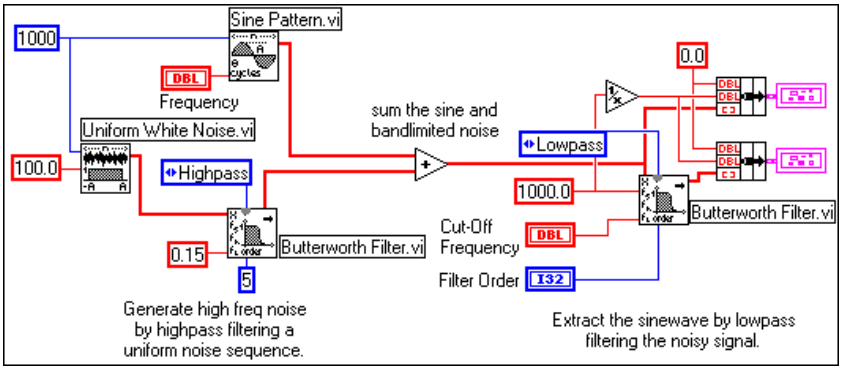


1. Open a new VI and build the front panel as shown above.
   a. Select a Digital Control from the **Numeric»Controls** palette and label it *Frequency.*
   b. Select Vertical Slide from the **Numeric»Controls** palette and label it *Cut-Off Frequency.*
   c. Select another Vertical Slide from the **Numeric»Controls** palette and label it *Filter Order.*

d. Select a Waveform Graph from the **Numeric»Graph** palette for displaying the noisy signal, and another Waveform Graph for displaying the original signal.

Block Diagram

2. Build the block diagram as shown below.



Sine Pattern VI (**Functions»Analysis»Signal Generation** palette) generates a sine wave of the desired frequency.

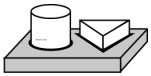
Uniform White Noise VI (**Functions»Analysis»Signal Generation** palette) generates uniform white noise that is added to the sinusoidal signal.

Butterworth Filter VI (**Functions»Analysis»Filters** palette) highpass filters the noise.

Notice that you are generating 10 cycles of the sine wave, and there are 1000 samples. Also, the sampling frequency to the **Butterworth Filter** VI on the right side is specified as 1000 Hz. Thus, effectively you are generating a 10 Hz signal.

3. Save the VI as `Extract the Sine Wave.vi` in the `LabVIEW\Activity` directory.
4. Switch back to the front panel. Select a **Frequency** of 10 Hz, a **Cut-Off Frequency** of 25 Hz, and a **Filter Order** of 5. Run the VI.
5. Reduce the **Filter Order** to 4, 3, and 2, and observe the difference in the filtered signal. Explain what happens as you lower the filter order.

6. When you finish, save the VI as Extract the Sine Wave.vi in the Dig.filt.llb library.
7. Close the VI.



End of Activity 16-1.

Summary

You have seen from the frequency response characteristics that practical filters differ from ideal filters. For practical filters, the gain in the passband may not always be equal to 1, the attenuation in the stopband may not always be -infinity, and there exists a transition region of finite width. The width of the transition region depends on the filter order, and the width decreases with increasing order.

You have also learned about both FIR and IIR digital filters. The output of FIR filters depends only on the current and past input values, whereas the output of IIR filters depends on the current and past input values as well as the past output values. You saw the frequency response of different designs of IIR filters and classified them according to the presence of ripples in the passband and/or the stopband. Because of the dependence of its output on past outputs, a transient appears at the output of an IIR filter each time the VI is called. This transient can be eliminated after the first call to the VI by setting its **init/cont** control to a TRUE value.

17

Curve Fitting

This chapter describes how to extract information from a data set to obtain a functional description. For examples of how to use the regression VIs, see the examples located in `examples\analysis\regressn.llb`.

Introduction to Curve Fitting

Curve fitting analysis is a technique for extracting a set of curve parameters or coefficients from the data set to obtain a functional description of the data set. The algorithm that fits a curve to a particular data set is known as the Least Squares Method and is discussed in most introductory textbooks in probability and statistics. The error is defined as

$$e(a) = [f(x,a) - y(x)]^2, \tag{17-1}$$

where $e(a)$ is the error, $y(x)$ is the observed data set, $f(x,a)$ is the functional description of the data set, and a is the set of curve coefficients which best describes the curve.

For example, let $a = \{a_0, a_1\}$. Then the functional description of a line is

$$f(x,a) = a_0 + a_1 x.$$

The least squares algorithm finds a by solving the system

$$\frac{\partial}{\partial a} e(a) = 0 \tag{17-2}$$

To solve this system, you set up and solve the Jacobian system generated by expanding Equation 17-2. After you solve the system for a, you can obtain an estimate of the observed data set for any value of x using the functional description $f(x, a)$.

In LabVIEW, the curve fitting VIs automatically set up and solve the Jacobian system and return the set of coefficients that best describes your data set. You can concentrate on the functional description of your data and not worry about solving the system in Equation 17-2.

Two input sequences, Y Values and X Values, represent the data set $y(x)$. A sample or point in the data set is

$$(x_i, y_i),$$

where x_i is the i^{th} element of the sequence X Values, and y_i is the i^{th} element of the sequence Y Values.

In general, for each predefined type of curve fit, there are two types of VIs, unless otherwise specified. One type returns only the coefficients, so that you can further manipulate the data. The other type returns the coefficients, the corresponding expected or fitted curve, and the mean squared error (MSE). Because it is a discrete system, the VI calculates the MSE, which is a relative measure of the residuals between the expected curve values and the actual observed values, using the formula

$$\text{MSE} = \frac{1}{n}\sum_{i=0}^{n-1}(f_i - y_i)^2 \qquad (17\text{-}3)$$

where f is the sequence representing the fitted values, y is the sequence representing the observed values, and n is the number of sample points observed.

The Analysis library offers both linear and nonlinear curve fitting algorithms. The different types of curve fitting in LabVIEW are outlined below:

- *Linear Fit*—fits experimental data to a straight line of the form $y = mx + c$.

$$y[i]=a_0+a_1*x[i]$$

- *Exponential Fit*—fits data to an exponential curve of the form $y = aexp(bx)$

$$y[i] = a_0*\exp(a_1*x[i])$$

- *General Polynomial Fit*—fits data to a polynomial function of the form

$$y = a + bx + cx\text{^}2 + \ldots$$
$$y[i] = a_0+a_1*x[i]+a_2*x[i]^2\ldots$$

- *General Linear Fit*—fits data to

$$y[i] = a_0 + a_1 * f_1(x[i]) + a_2 * f_2(x[i]) + \ldots$$

where y[i] is a linear combination of the parameters a_0, a_1, a_2.... The general linear fit also features selectable algorithms for better precision and accuracy. For example, $y = a_0 + a_1 * \sin(x)$ is a linear fit because *y* has a linear relationship with parameters a_0 and a_1. Polynomial fits are always linear fits for the same reason. But special algorithms can be designed for the polynomial fit to speed up the fitting processing and improve accuracy.

- *Nonlinear Levenberg-Marquardt Fit*—fits data to

$$y[i] = f(x[i], a_0, a_1, a_2 \ldots)$$

where a_0, a_1, a_2... are the parameters. This method is the most general method and does not require *y* to have a linear relationship with a_0, a_1, a_2.... It can be used to fit linear or nonlinear curves, but is almost always used to fit a nonlinear curve, because the general linear fit method is better suited to linear curve fitting. The Levenberg-Marquardt method does not always guarantee a correct result, so it is absolutely necessary to verify the results.

Applications of Curve Fitting

The practical applications of curve fitting are numerous. Some of them are listed below.

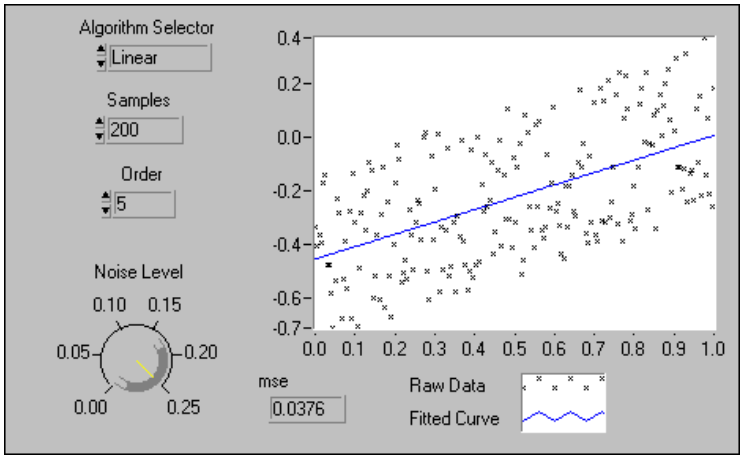
- Removal of measurement noise.
- Filling in missing data points (for example, if one or more measurements were missed or improperly recorded).
- Interpolation (estimation of data between data points; for example, if the time between measurements is not small enough).
- Extrapolation (estimation of data beyond data points; for example, if you are looking for data values before or after the measurements were taken).
- Differentiation of digital data. (For example, if you need to find the derivative of the data points. The discrete data can be modeled by a polynomial, and the resulting polynomial equation can be differentiated.)
- Integration of digital data (for example, to find the area under a curve when you have only the discrete points of the curve).
- To obtain the trajectory of an object based on discrete measurements of its velocity (first derivative) or acceleration (second derivative).

Activity 17-1. Use the Curve Fitting VIs

Your objective is to use and compare the Linear, Exponential and Polynomial Curve Fit VIs to obtain the set of least square coefficients that best represent a set of data points.

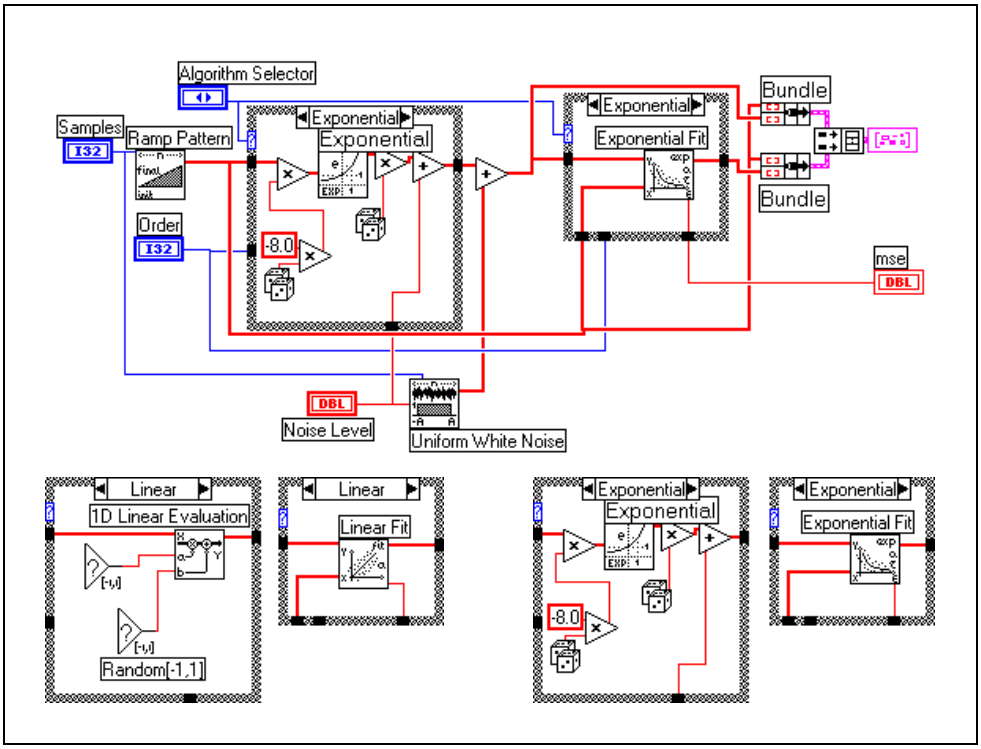
Front Panel

1. Open the Regressions Demo VI from the `library regressn.llb`. The front panel and block diagram are already built for you.



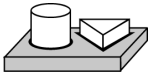
This VI generates “noisy” data samples that are approximately linear, exponential, or polynomial. It then uses the corresponding analysis curve fitting VIs to determine the parameters of the curve that best fits those data points. (At this stage, you do not need to worry about how the noisy data samples are generated.) You can control the noise amplitude with the **Noise Level** control on the front panel.

Block Diagram



2. Select *Linear* in the **Algorithm Selector** control, and set the **Noise Level** control to about 0.1. Run the VI. Notice the spread of the data points and the fitted curve (straight line).
3. Experiment with different values of **Order** and **Noise Level**. What do you notice? How does the mse change?
4. Change the **Algorithm Selector** to *Exponential* and run the VI. Experiment with different values of **Order** and **Noise Level**. What do you notice?
5. Change the **Algorithm Selector** to *Polynomial* and run the VI. Experiment with different values of **Order** and **Noise Level**. What do you notice?
6. In particular, with the **Algorithm Selector** control set to *Polynomial*, change the **Order** to 0 and run the VI. Then change it to 1 and run the VI. Explain your observations.

7. Depending on your observations in steps 2, 3, 4, and 5, for which of the algorithms (Linear, Exponential, Polynomial) is the **Order** control the most effective? Why?
8. Close the VI and quit. Do not save any changes.



End of Activity 17-1.

General LS Linear Fit Theory

The General LS Linear Fit Problem can be described as follows.

Given a set of observation data, find a set of coefficients that fit the linear "model."

$$y_i = b_o x_{i0} + \ldots + b_{k-1} x_{ik-1}$$

$$= \sum_{j=0}^{k-1} b_j x_{ij} \qquad \text{i=0, 1, ... , n–1,} \tag{17-4}$$

where *B* is the set of **Coefficients**, *n* is the number of elements in **Y Values** and the number of rows of **H**, and *k* is the number of **Coefficients**.

x_{ij} is your observation data, which is contained in **H**.

$$H = \begin{bmatrix} x_{00} & x_{01} \cdots & x_{0k-1} \\ x_{10} & x_{11} \cdots & x_{1k-1} \\ \vdots & & \\ x_{n-10} & x_{n-12} \cdots & x_{n-1k-1} \end{bmatrix}$$

Equation 17-4 can also be written as $Y = HB$.

This is a multiple linear regression model, which uses several variables $x_{i0}, x_{i1}, \ldots, x_{ik-1}$, to predict one variable y_i. In contrast, the Linear Fit, Exponential Fit, and Polynomial Fit VIs are all based on a single predictor variable, which uses one variable to predict another variable.

In most cases, we have more observation data than coefficients. The equations in 17-4 may not have the solution. The fit problem becomes to find the coefficient *B* that minimizes the difference between the observed data, y_i and the predicted value:

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij} .$$

This VI uses the least chi-square plane method to obtain the coefficients in 17-4, that is, finding the solution, *B*, which minimizes the quantity:

$$\chi^2 = \sum_{i=0}^{n-1} \left(\frac{y_i - z_i}{\sigma_i} \right)^2 = \sum_{i=0}^{n-1} \left(\frac{y_i - \sum_{j=0}^{k-1} b_j x_{ij}}{\sigma_i} \right)^2 = |H_0 B - Y_0|^2 \qquad (17\text{-}5)$$

where

$$h_{oij} = \frac{x_{ij}}{\sigma_i}, \; y_{oi} = \frac{y_i}{\sigma_i}, \; i=0, 1, \ldots, n-1; \; j=0, 1, \ldots, k-1.$$

In this equation, σ_i is the **Standard Deviation**. If the measurement errors are independent and normally distributed with constant standard deviation $\sigma_i = \sigma$, the preceding equation is also the least square estimation.

There are different ways to minimize χ^2. One way to minimize χ^2 is to set the partial derivatives of χ^2 to zero with respect to b_0, b_1, ... , b_{k-1}.

$$\begin{cases} \frac{\partial \chi^2}{\partial b_0} = 0 \\ \frac{\partial \chi^2}{\partial b_1} = 0 \\ \vdots \\ \frac{\partial \chi^2}{\partial b_{k-1}} = 0 \end{cases}$$

The preceding equations can be derived to:

$$H_0^T H_0 B = H_0^T Y \tag{17-6}$$

Where H_0^T is the transpose of $\mathbf{H}_0$.

The equations in 17-6 are also called normal equations of the least-square problems. You can solve them using LU or Cholesky factorization algorithms, but the solution from the normal equations is susceptible to roundoff error.

An alternative, and preferred way to minimize χ^2 is to find the least-square solution of equations

$$H_0 B = Y_0.$$

You can use QR or SVD factorization to find the solution, *B*. For QR factorization, you can choose Householder, Givens, and Givens2 (also called fast Givens).

Different algorithms can give you different precision, and in some cases, if one algorithm cannot solve the equation, perhaps another algorithm can. You can try different algorithms to find the best one based on your observation data.

The Covariance matrix C is computed as

$$C = (H_0^T H_0)^{-1}.$$

The Best Fit Z is given by

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij}$$

The mse is obtained using the following formula:

$$\boldsymbol{mse} = \frac{1}{n} \sum_{i=0}^{n-1} \left(\frac{y_i - z_i}{\sigma_i} \right)^2$$

The polynomial fit that has a single predictor variable can be thought of as a special case of multiple regression. If the observation data sets are $\{x_i, y_i\}$ where $i = 0, 1, \ldots, n-1$, the model for polynomial fit is

$$y_i = \sum_{j=0}^{k-1} b_j x_i^j = b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_{k-1} x_i^{k-1} \qquad (17\text{-}7)$$

$$i = 0, 1, 2, \ldots, n-1.$$

Comparing equations 17-4 and 17-7 shows that $x_{ij} = x_i^j$. In other words,

$$x_{i0} = x_i^0, \quad x_{i1} = x_i, \quad x_{i2} = x_i^2, \ldots \quad x_{ik-1} = x_i^{k-1}$$
$$= 1$$

In this case, you can build **H** as follows:

$$H = \begin{Bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{k-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{k-1} \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{k-1} \end{Bmatrix}$$

Instead of using $x_{ij} = x_j^i$, you can also choose another function formula to fit the data sets $\{x_i, y_i\}$. In general, you can select $x_{ij} = f_j(x_i)$. Here, $f_j(x_i)$ is the function model that you choose to fit your observation data. In polynomial fit, $f_j(x_i) = x_i^j$.

In general, you can build **H** as follows:

$$H = \begin{Bmatrix} f_0(x_0) & f_1(x_0) & f_2(x_0) & \dots & f_{k-1}(x_0) \\ f_0(x_1) & f_1(x_1) & f_2(x_1) & \dots & f_{k-1}(x_1) \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ f_0(x_{n-1}) & f_1(x_{n-1}) & f_2(x_{n-1}) & \dots & f_{k-1}(x_{n-1}) \end{Bmatrix}$$

Your fit model is:

$$y_i = b_0 f_0(x) + b_1 f_1(x) + \dots + b_{k-1} f_{k-1}(x) .$$

How to Use the General LS Linear Fit VI

The Linear Fit VI calculates the coefficients a_0 and a_1 that best fits the experimental data (x[i] and y[i]) to a straight line model given by

$$y[i] = a_0 + a_1*x[i]$$

Here, y[i] is a linear combination of the coefficients a_0 and a_1. You can extend this concept further so that the multiplier for a1 is some function of *x*. For example:

$$y[i] = a_0 + a_1*\sin(\omega x[i])$$

or

$$y[i] = a_0 + a_1*x[i]^2$$

or

$$y[i] = a_0 + a_1*\cos(\omega x[i]^2)$$

where ω is the angular frequency. In each of these cases, y[i] is a linear combination of the coefficients a_0 and a_1. This is the basic idea behind the General LS Linear Fit VI, where the y[i] can be linear combinations of several coefficients, each of which may be multiplied by some function of the x[i]. Therefore, you can use it to calculate coefficients of the functional models that can be represented as linear combinations of the coefficients, such as

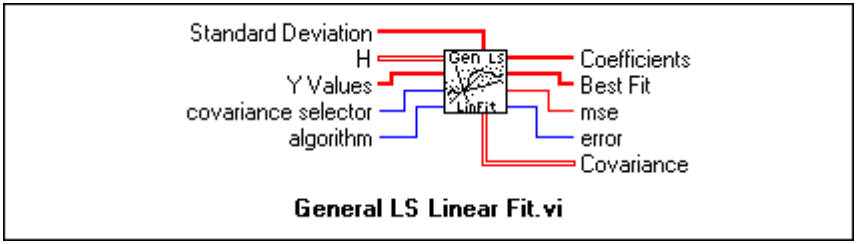
$$y = a_0 + a_1*\sin(\omega x)$$

or

$$y = a_0 + a_1*x^2 + a2*\cos(\omega x^2)$$
$$y = a_0 + a_1*(3\sin(\omega x)) + a_2*x^3 + a_3 / x + ...$$

In each case, note that *y* is a *linear* function of the coefficients (although it may be a nonlinear function of *x*).

You will now see how to use the General LS Linear Fit VI to find the best linear fit to a set of data points. The inputs and outputs of the General LS Linear Fit VI are shown below.



The data that you collect (x[i] and y[i]) is to be given to the inputs **H** and **Y Values**. The **Covariance** output is the matrix of covariances between the coefficients a_k, where c_{ij} is the covariance between a_i and a_j, and c_{kk} is the variance of a_k. At this stage, you need not be concerned about the inputs **Standard Deviation**, **covariance selector,** and **algorithm**. For now, you will just use their default values. You can refer to the *Analysis Online Reference* for more details on these inputs.

The matrix **H** is known as the *Observation Matrix* and will be explained in more detail later. **Y Values** is the set of observed data points y[i]. For example, suppose you have collected samples (**Y Values**) from a transducer and you want to solve for the coefficients of the model:

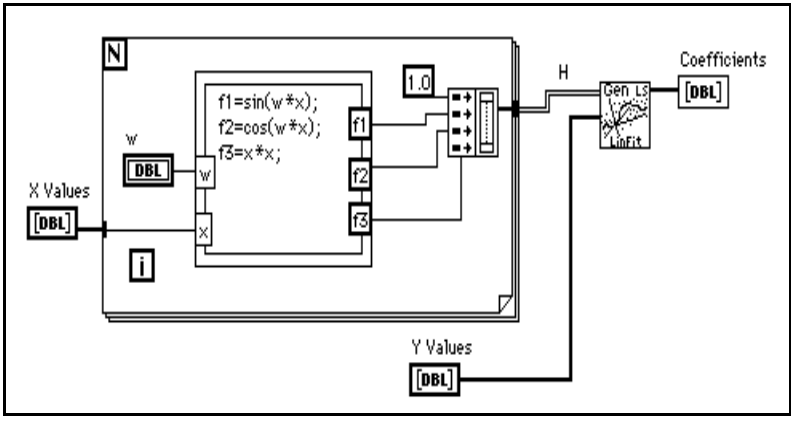
$$y = a_o + a_1 \sin(\omega x) + a_2 \cos(\omega x) + a_3 x^2$$

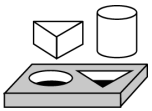
You see that the multiplier for each a_j (0 ð j ð3) is a different function. For example, a_0 is multiplied by 1, a_1 is multiplied by sin(ωx), a_2 is multiplied by cos(ωx), and so on. To build **H**, you set each column of **H** to the independent functions evaluated at each x value, x[i]. Assuming there are 100 "x" values, **H** would be:

$$H = \begin{bmatrix} 1 & \sin(\omega x_0) & \cos(\omega x_0) & x_0^2 \\ 1 & \sin(\omega x_1) & \cos(\omega x_1) & x_1^2 \\ 1 & \sin(\omega x_2) & \cos(\omega x_2) & x_2^2 \\ \cdots & \cdots & \cdots & \cdots \\ 1 & \sin(\omega x_{99}) & \cos(\omega x_{99}) & x_{99}^2 \end{bmatrix}$$

If you have *N* data points and *k* coefficients (a_0, a_1, ...a_{k-1}) for which to solve, **H** will be an *N-by-k* matrix with *N* rows and *k* columns. Thus, the number of rows of **H** is equal to the number of elements in **Y Values**, whereas the number of columns of **H** is equal to the number of coefficients for which you are trying to solve.

In practice, **H** is not available and must be built. Given that you have the *N* independent **X Values** and observed **Y Values**, the following block diagram demonstrates how to build **H** and use the General LS Linear Fit VI.





Activity 17-2. Use the General LS Linear Fit VI

For this activity, your objective is to learn how to set up the input parameters and use the General LS Linear Fit VI.

This activity demonstrates how to use the General LS Linear Fit VI to obtain the set of least square coefficients *a* and the fitted values, and also how to set up the input parameters to the VI.

The purpose is to find the set of least square coefficients **a** that best represent the set of data points (x[i], y[i]). As an example, suppose that we have a physical process that generates data using the relationship

$$y = 2h_0(x) + 3h_1(x) + 4h_2(x) + \text{noise} \qquad (17\text{-}8)$$

where

$$h_0(x) = \sin(x^2),$$
$$h_1(x) = \cos(x),$$
$$h_2(x) = \frac{1}{x+1},$$

and *noise* is a random value. Also, assume you have some idea of the general form of the relationship between *x* and *y*, but are not quite sure of the coefficient values. So, you may think that the relationship between *x* and *y* is of the form

$$y = a_0f_0(x) + a_1f_1(x) + a_2f_2(x) + a_3f_3(x) + a_4f_4(x) \qquad (17\text{-}9)$$

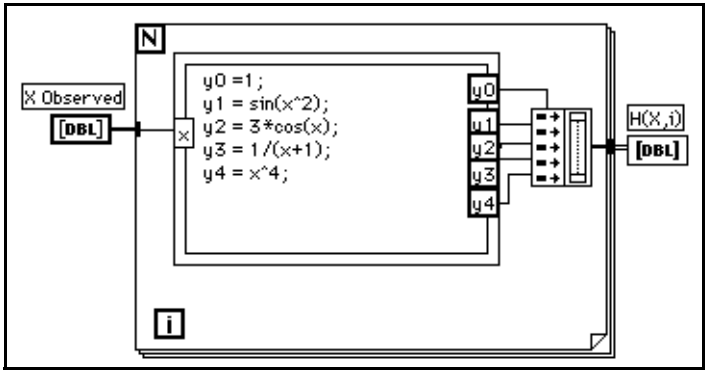
where

$$f_o(x) = 1.0,$$
$$f_1(x) = \sin(x^2),$$
$$f_2(x) = 3\cos(x),$$
$$f_3(x) = \frac{1}{x+1},$$
$$f_4(x) = x^4.$$

Equations 17-8 and 17-9 respectively correspond to the actual physical process and to your guess of this process. The coefficients you choose in your guess may be close to the actual values, or may be far away from them. Your objective now is to accurately determine the coefficients *a*.

Building the Observation Matrix

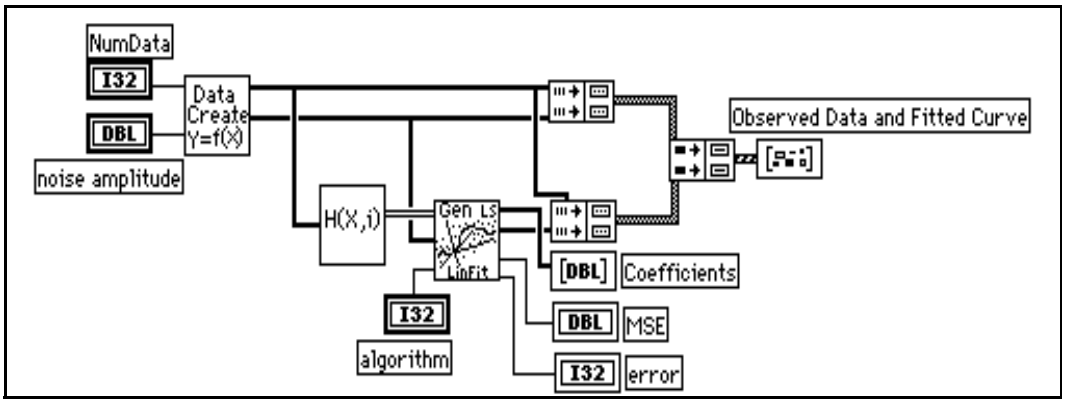
To obtain the coefficients *a*, you must supply the set of (x[i], y[i]) points in the arrays **H** and **Y Values** (where the matrix **H** is a 2D array) to the General LS Linear Fit VI. The x[i] and y[i] points are the values observed in your experiment. A simple way to build the matrix **H** is to use the Formula Node as shown in the following block diagram.



You can edit the formula node to change, add, or delete functions. At this point, you have all the necessary inputs to use the General LS Linear Fit VI to solve for *a*. To obtain equation (1) from equation (2), you need to multiply $f_0(x)$ by 0.0, $f_1(x)$ by 2.0, $f_2(x)$ by 1.0, $f_3(x)$ by 4.0 and $f_4(x)$ by 0.0. Thus, looking at equations (1) and (2), note that the expected set of coefficients are

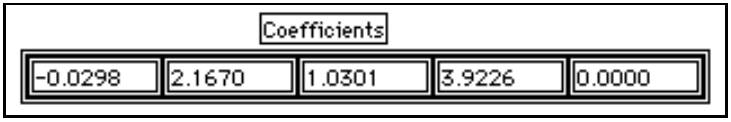
$$a = \{0.0, 2.0, 1.0, 4.0, 0.0\} \quad .$$

The block diagram below demonstrates how to set up the General LS Linear Fit VI to obtain the coefficients and a new set of *y* values.



The subVI labeled Data Create generates the **X** and **Y** arrays. You can replace this icon with one that actually collects the data in your experiments. The icon labeled **H(X,i)** generates the 2D matrix **H**.

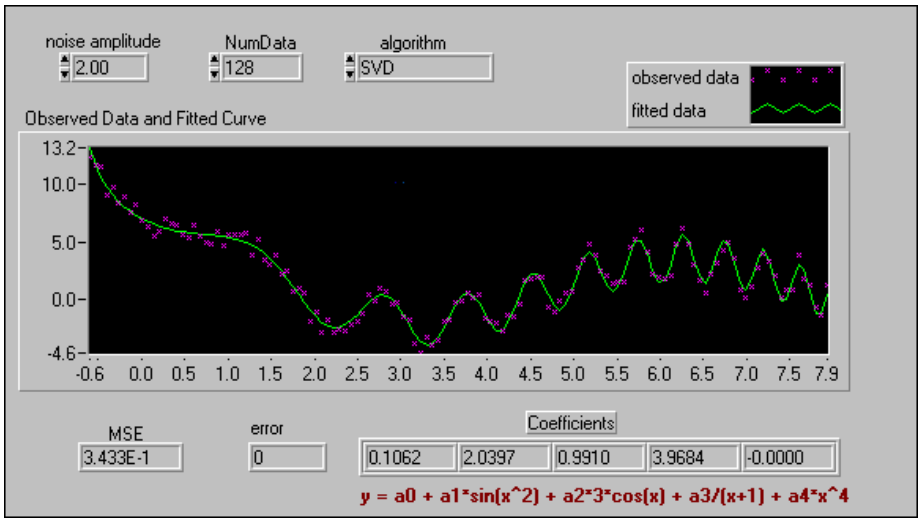
The last portion of the block diagram overlays the original and the estimated data points and produces a visual record of the General LS Linear Fit. Executing the General LS Linear Fit VI with the values of **X**, **Y**, and **H** returns the following set of coefficients.



The resulting equation is thus

$$y = 0.0298(1) + 2.1670\sin(x^2) + 1.0301(3\cos(x)) + 3.9226/(x+1) + 0.00(x^4)$$
$$= 0.0298 + 2.1670\sin(x^2) + 1.0301(3\cos(x)) + 3.9226/(x+1)$$

The following graph displays the results.



You will now see the VI in which this particular example has been implemented.

1. Open the General LS Fit Example VI from the library `examples\analysis\regressn.llb`.
2. Examine the block diagram. Make sure you understand it.
3. Examine the front panel.

   noise amplitude: can change the amplitude of the noise added to the data points. The larger this value, the more the spread of the data points.

   NumData: the number of data points that you want to generate.

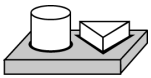
   algorithm: provides a choice of six different algorithms to obtain the set of coefficients and the fitted values. In this particular example, there is no significant difference among different algorithms. You can select different algorithms from the front panel to see the results. In some cases, different algorithms may have significant differences, depending on your observed data set.

   MSE: gives the mean squared error. The smaller the MSE, the better the fit.

   error: gives the error code in case of any errors. If error code = 0, it indicates no error. For a list of error codes, see Appendix A, *Error Codes*, in the *LabVIEW Function and VI Reference Manual*.

   Coefficients: the calculated values of the coefficients (a_0, a_1, a_2, a_3, and a_4) of the model.

4. Run the VI with progressively larger values of the **noise amplitude**. What happens to the observed data plotted on the graph? What about the MSE?
5. For a fixed value of **noise amplitude**, run the VI by choosing different algorithms from the **algorithm** control. Do you find that any one algorithm is better than the other? Which one gives you the lowest MSE?
6. When you finish, close the VI. Do not save any changes.



End of Activity 17-2.

Nonlinear Lev-Mar Fit Theory

This VI determines the set of coefficients that minimize the chi-square quantity:

$$\chi^2 = \sum_{i=0}^{N-1} \left(\frac{y_i - f(x_i; a_1 \ldots a_M)}{\sigma_i} \right)^2 \qquad (17\text{-}10)$$

In this equation, (x_i, y_i) are the input data points, and $f(x_i;a_1...a_M) = f(X, A)$ is the nonlinear function where $a_1...a_M$ are coefficients. If the measurement errors are independent and normally distributed with constant, standard deviation $\sigma_i = \sigma$, this is also the least-square estimation.

You must specify the nonlinear function $f = f(X, A)$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, which is a subVI of the Nonlinear Lev-Mar Fit VI. You can access the Target Fnc & Deriv NonLin VI by selecting it from the menu that appears when you select **Project»This VI's SubVIs**.

This VI provides two ways to calculate the Jacobian (partial derivatives with respect to the coefficients) needed in the algorithm. These two methods follow:

- Numerical calculation—Uses a numerical approximation to compute the Jacobian.
- Formula calculation—Uses a formula to compute the Jacobian. You need to specify the Jacobian function $\partial f / \partial A$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, as well as

the nonlinear function $f = f(X, A)$. This is a more efficient computation than the numerical calculation, because it does not require a numerical approximation to the Jacobian.

The input arrays **X** and **Y** define the set of input data points. The VI assumes that you have prior knowledge of the nonlinear relationship between the x and y coordinates. That is, $f = f(X, A)$, where the set of coefficients, A, is determined by the Levenberg-Marquardt algorithm.

Using this function successfully sometimes depends on how close your initial guess coefficients are to the solution. Therefore, it is always worth taking effort and time to obtain good initial guess coefficients to the solution from any available resources before using the function.

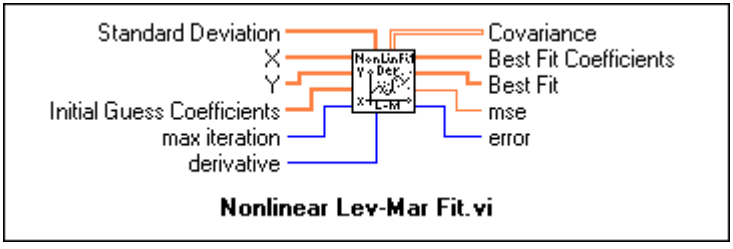
Using the Nonlinear Lev-Mar Fit VI

So far, you have seen VIs that are used when there is a linear relationship between y and the coefficients a_0, a_1, a_2, ... However, when a nonlinear relationship exists, you can use the Nonlinear Lev-Mar Fit VI to determine the coefficients. This VI uses the Levenberg-Marquardt method, which is very robust, to find the coefficients $\mathbf{A} = \{a_0, a_1, a_2, ..., a_k\}$ of the nonlinear relationship between **A** and y[i]. The VI assumes that you have prior knowledge of the nonlinear relationship between the x and y coordinates.

As a preliminary step, you need to specify the nonlinear function in the Formula Node on the block diagram of one of the subVIs of the Nonlinear Lev-Mar Fit VI. This particular subVI is the Target Fnc and Deriv NonLin VI. You can access the Target Fnc and Deriv NonLin VI by selecting it from the menu that appears when you select **Project»This VI's SubVIs**.

Note ***When using the Nonlinear Lev-Mar Fit VI, you also must specify the nonlinear function in the Formula Node on the block diagram of the Target Fnc and Deriv NonLin VI.***

The connections to the Nonlinear Lev-Mar Fit VI are shown below:

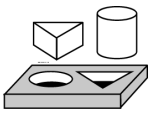


X and **Y** are the input data points x[i] and y[i].

Initial Guess Coefficients is your initial guess as to what the coefficient values are. The coefficients are those used in the formula that you entered in the **Formula Node** of the **Target Fnc and Deriv NonLin** VI. Using the **Nonlinear Lev-Mar Fit** VI successfully sometimes depends on how close your initial guess coefficients are to the actual solution. Therefore, it is always worth taking the time and effort to obtain a good initial guess to the solution from any available resource.

For now, you can leave the other inputs to their default values. For more information on these inputs, see the *Analysis Online Reference*.

Best Fit Coefficients: the values of the coefficients (a_0, a_1, ...) that best fit the model of the experimental data.



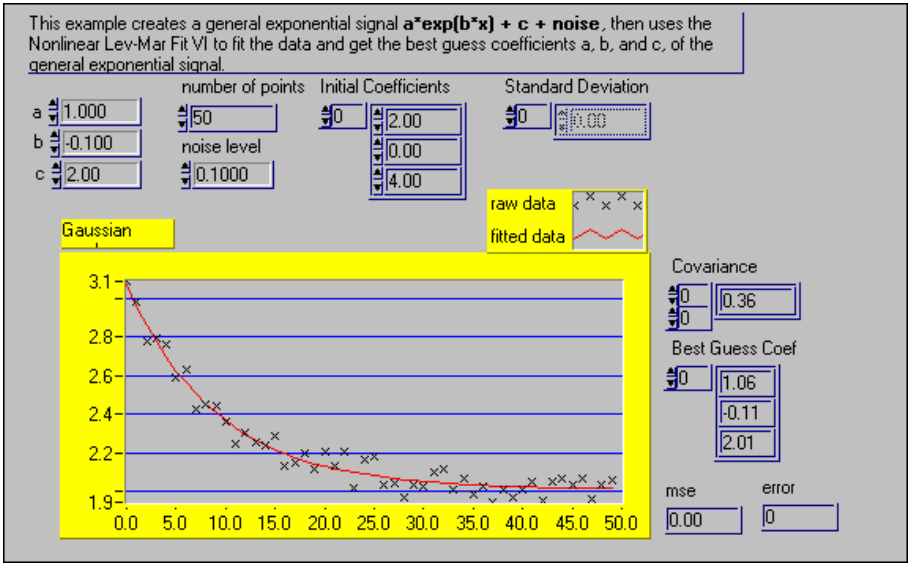
Activity 17-3. Use the Nonlinear Lev-Mar Fit VI

For this activity, your objective is to create a general exponential signal a*exp(b*x) + c + noise, and then use the Nonlinear Lev-Mar Fit VI to fit the data and get the best guess coefficients a, b, and c of the general exponential signal.

In this activity, you will see how to use the Nonlinear Lev-Mar Fit VI to determine the coefficients *a, b,* and *c,* of a nonlinear function given by a*exp(b*x) + c.

Front Panel

1. Open the Nonlinear Lev-Mar Exponential Fit VI from the library `examples\analysis\regressn.llb`**.** The front panel is shown in the following illustration.



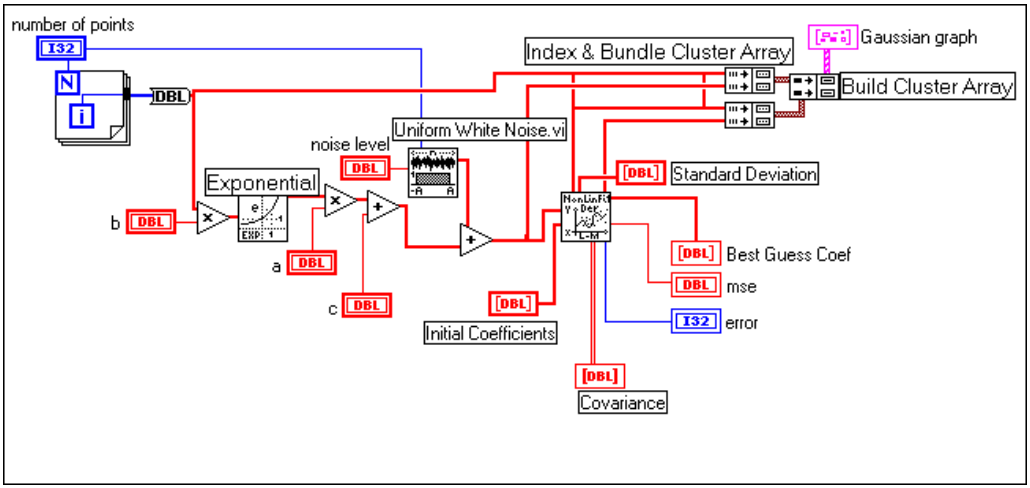
The **a**, **b**, and **c** controls determine the actual values of the coefficients *a*, *b*, and *c*. The **Initial Coefficients** control is your educated guess as to the actual values of *a*, *b*, and *c*. Finally, the **Best Guess Coef** indicator gives you the values of *a*, *b*, and *c* calculated by the **Nonlinear Lev-Mar Fit** VI. To simulate a more practical example, we also add noise to this equation, thus making it of the form:

$$a*\exp(b*x) + c + \text{noise}$$

The **noise level** control adjusts the noise level. Note that the actual values of *a*, *b*, and *c* being chosen are +1.0, –0.1 and 2.0. In the **Initial Coefficients** control, the default guess for these is $a = 2.0$, $b = 0$, and $c = 4.0$.

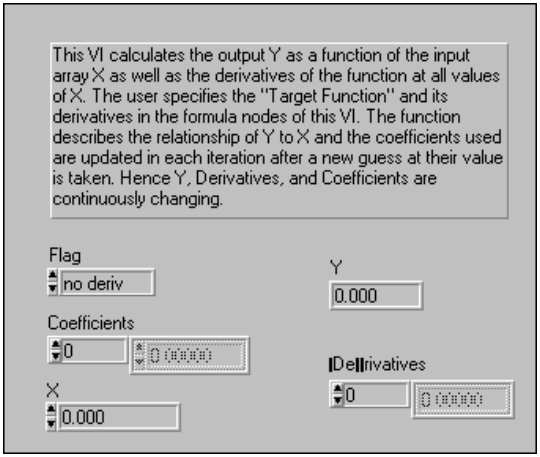
Block Diagram

2. Examine the block diagram.

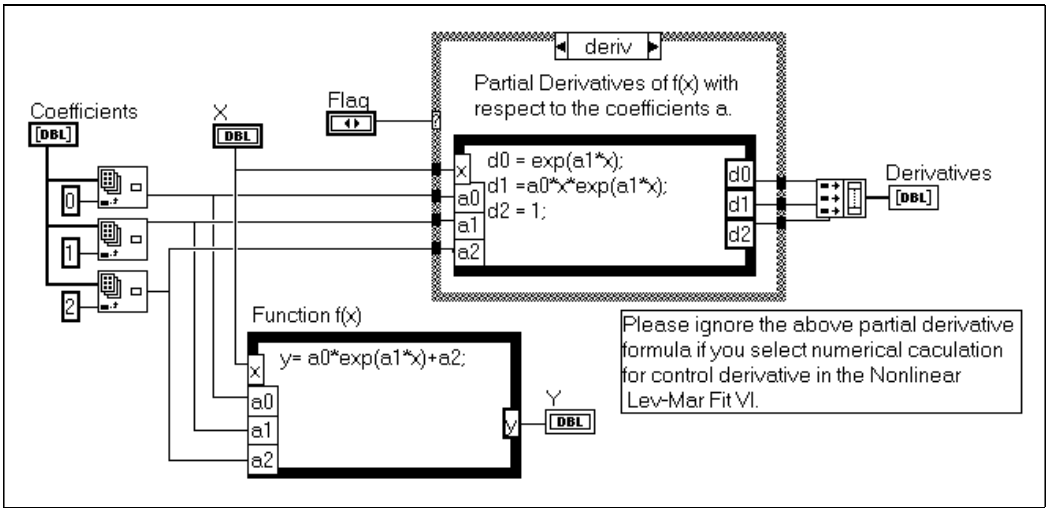


The data samples of the exponential function are simulated using the Exponential VI (**Numeric»Logarithmic** subpalette) and uniform white noise is added to the samples with the help of the Uniform White Noise VI (**Analysis»Signal Generation** subpalette).

3. From the **Project** menu, select **Unopened SubVIs»Target Fnc and Deriv NonLin** VI. The front panel of the Target Fnc and Deriv NonLin VI opens, as shown below.

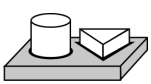


4. Switch to the block diagram.



Observe the Formula Node at the bottom. It has the form of the function whose parameters (a0, a1, and a2) you are trying to evaluate.

5. Close the front panel and the block diagram of the Target Fnc and Deriv NonLin VI.
6. Run the NonLinear Lev-Mar Exponential Fit VI. Note that the values of the coefficients returned in **Best Guess Coef** are very close to the actual values entered in the **Initial Coefficients** control. Also note the value of the **mse**.
7. Increase the **noise level** from 0.1 to 0.5. What happens to the **mse** and the coefficient values in **Best Guess Coef**? Why?
8. Change the **noise level** back to 0.1 and the **Initial Coefficients** to 5.0, –2.0, and 10.0, and run the VI. Note the values returned in the **Best Guess Coef** and the **mse** indicators.
9. With the **noise level** still at 0.1, change your guess of the **Initial Coefficients** to 5.0, 8.0, and 10.0, and run the VI. This time, your guess is further away than the one you chose in step 4. Notice the error. This illustrates how important it is to have a reasonably educated guess for the coefficients.
10. When you finish, close the VI. Do not save any changes.



End of Activity 17-3.

18

Linear Algebra

This chapter explains how to use the linear algebra VIs to perform matrix computation and analysis. For examples of how to use the linear algebra VIs, see the examples located in `examples\analysis\linxmpl.llb`.

Linear Systems and Matrix Analysis

Systems of linear algebraic equations arise in many applications that involve scientific computations such as signal processing, computational fluid dynamics, and others. Such systems may occur naturally or may be the result of approximating differential equations by algebraic equations.

Types of Matrices

Whatever the application, it is always necessary to find an accurate solution for the system of equations in a very efficient way. In matrix-vector notation, such a system of linear algebraic equations has the form

$$Ax = b$$

where A is an $n \times n$ matrix, b is a given vector consisting of n elements, and x is the unknown solution vector to be determined. A matrix is represented by a 2D array of elements. These elements may be real numbers, complex numbers, functions, or operators. The matrix A shown below is an array of m rows and n columns with $m \times n$ elements.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

Here, $a_{i,j}$ denotes the $(i,j)^{th}$ element located in the i^{th} row and the j^{th} column. In general, such a matrix is called a *rectangular matrix*. When $m = n$, so that the number of rows is equal to the number of columns, it is called a *square matrix*. An $m \times 1$ matrix (m rows and one column) is called a *column vector*. A *row vector* is a $1 \times n$ matrix (1 row and n columns). If all

the elements other than the diagonal elements are zero (that is, $a_{i,j} = 0$, $i \neq j$), such a matrix is called a *diagonal matrix*. For example,

$$A = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

is a diagonal matrix. A diagonal matrix with all the diagonal elements equal to one is called an *identity matrix*, also known as *unit matrix*. If all the elements below the main diagonal are zero, then the matrix is known as an *upper triangular matrix*. On the other hand, if all the elements above the main diagonal are zero, then the matrix is known as a *lower triangular matrix*. When all the elements are real numbers, the matrix is referred to as a *real matrix*. On the other hand, when at least one of the elements of the matrix is a complex number, the matrix is referred to as a *complex matrix*. To make things simpler to understand, you will work mainly with real matrices in this lesson. However, for the adventurous, there are also some activities involving complex matrices.

Determinant of a Matrix

One of the most important attributes of a matrix is its *determinant*. In the simplest case, the determinant of a 2 x 2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

is given by $ad - bc$. The determinant of a square matrix is formed by taking the determinant of its elements. For example, if

$$A = \begin{bmatrix} 2 & 5 & 3 \\ 6 & 1 & 7 \\ 1 & 6 & 9 \end{bmatrix}$$

then the determinant of **A**, denoted by $|A|$, is

$$|A| = \left|\begin{bmatrix} 2 & 5 & 3 \\ 6 & 1 & 7 \\ 1 & 6 & 9 \end{bmatrix}\right| = \left(2\left|\begin{bmatrix} 1 & 7 \\ 6 & 9 \end{bmatrix}\right| - 5\left|\begin{bmatrix} 6 & 7 \\ 1 & 9 \end{bmatrix}\right| + 3\left|\begin{bmatrix} 6 & 1 \\ 1 & 6 \end{bmatrix}\right|\right) =$$

$$2(-33) - 5(47) + 3(35) = -196$$

The determinant tells many important properties of the matrix. For example, if the determinant of the matrix is zero, then the matrix is *singular*. In other words, the above matrix (with nonzero determinant) is *nonsingular*. You will revisit the concept of singularity later in the section *Matrix Inverse and Solving Systems of Linear Equations*, when the lesson discusses the solution of linear equations and matrix inverses.

Transpose of a Matrix

The *transpose* of a real matrix is formed by interchanging its rows and columns. If the matrix B represents the transpose of A, denoted by A^T, then $b_{j,i}=a_{i,j}$. For the matrix A defined above,

$$B = A^T = \begin{bmatrix} 2 & 6 & 1 \\ 5 & 1 & 6 \\ 3 & 7 & 9 \end{bmatrix}$$

In case of complex matrices, we define complex conjugate transposition. If the matrix D represents the *complex conjugate transpose* (if $a = x + iy$, then complex conjugate $a^* = x - iy$) of a complex matrix C, then

$$D = C^H \Rightarrow d_{i,j} = c^*_{j,i}$$

That is, the matrix D is obtained by replacing every element in C by its complex conjugate and then interchanging the rows and columns of the resulting matrix.

A real matrix is called a *symmetric matrix* if the transpose of the matrix is equal to the matrix itself. The example matrix *A* is not a symmetric matrix. If a complex matrix C satisfies the relation $C = C^H$, then *C* is called a *Hermitian matrix*.

Can You Obtain One Vector as a Linear Combination of Other Vectors? (Linear Independence)

A set of vectors $x_1, x_2, ..., x_n$ is said to be *linearly dependent* if and only if there exist scalars $\alpha_1, \alpha_2, ..., \alpha_n$, not all zero, such that

$$\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n = 0$$

In simpler terms, if one of the vectors can be written in terms of a linear combination of the others, then the vectors are said to be linearly dependent.

If the only set of α_i for which the above equation holds is $\alpha_1 = 0$, $\alpha_2 = 0, ..., \alpha_n = 0$, then the set of vectors $x_1, x_2, ..., x_n$ is said to be *linearly independent*. So, in this case, none of the vectors can be written in terms of a linear combination of the others. Given any set of vectors, the above equation always holds for $\alpha_1 = 0$, $\alpha_2 = 0, ..., \alpha_n = 0$. Therefore, to show the linear independence of the set, you must show that $\alpha_1 = 0$, $\alpha_2 = 0, ..., \alpha_n = 0$ is the only set of α_i for which the above equation holds.

For example, first consider the vectors

$$x = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad y = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Notice that $\alpha_1 = 0$ and $\alpha_2 = 0$ are the only values, for which the relation $\alpha_1 x + \alpha_2 y = 0$ holds true. Hence, these two vectors are linearly independent of each other. Let us now look at vectors

$$x = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad y = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Notice that, if $\alpha_1 = -2$ and $\alpha_2 = 1$, then $\alpha_1 x + \alpha_2 y = 0$. Therefore, these two vectors are linearly dependent on each other. You must completely understand this definition of linear independence of vectors to fully appreciate the concept of the *rank* of the matrix as discussed next.

How Can You Determine Linear Independence? (Matrix Rank)

The *rank* of a matrix *A*, denoted by $\rho(A)$, is the maximum number of linearly independent columns in *A*. If you look at the example matrix *A*, you will find that all the columns of *A* are linearly independent of each other. That is, none of the columns can be obtained by forming a linear combination of the other columns. Hence, the rank of the matrix is 3. Consider one more example matrix, *B*, where

$$B = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 2 \end{bmatrix}$$

This matrix has only two linearly independent columns, because the third column of *B* is linearly dependent on the first two columns. Hence, the rank of this matrix is 2. It can be shown that the number of linearly independent columns of a matrix is equal to the number of independent rows. So, the rank can never be greater than the smaller dimension of the matrix. Consequently, if *A* is an $n \times m$ matrix, then

$$\rho(A) \leq min(n, m)$$

where *min* denotes the minimum of the two numbers. In matrix theory, the rank of a square matrix pertains to the highest order nonsingular matrix that can be formed from it. Remember from the earlier discussion that a matrix is singular if its determinant is zero. So, the rank pertains to the highest order matrix that you can obtain whose determinant is not zero. For example, consider a 4 x 4 matrix

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 0 & 2 \end{bmatrix}$$

For this matrix, $det(B) = 0$, but

$$\left| \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{bmatrix} \right| = -1$$

Hence, the rank of *B* is 3. A square matrix has full rank if and only if its determinant is different from zero. Matrix *B* is not a full-rank matrix.

"Magnitude" (Norms) of Matrices

You must develop a notion of the "magnitude" of vectors and matrices to measure errors and sensitivity in solving a linear system of equations. As an example, these linear systems can be obtained from applications in control systems and computational fluid dynamics. In two dimensions, for example, you cannot compare two vectors $\boldsymbol{x} = \begin{bmatrix} x1 & x2 \end{bmatrix}$ and $y = \begin{bmatrix} y1 & y2 \end{bmatrix}$, because you might have $x1 > y1$ but $x2 < y2$. A vector norm is a way to assign a scalar quantity to these vectors so that they can be compared with each other. It is similar to the concept of magnitude, modulus, or absolute value for scalar numbers.

There are ways to compute the norm of a matrix. These include the *2-norm* (Euclidean norm), the *1-norm*, the *Frobenius norm* (F-norm), and the *Infinity norm* (inf-norm). Each norm has its own physical interpretation. Consider a unit ball containing the origin. The Euclidean norm of a vector is simply the factor by which the ball must be expanded or shrunk in order to encompass the given vector exactly. This is shown in the figures below:

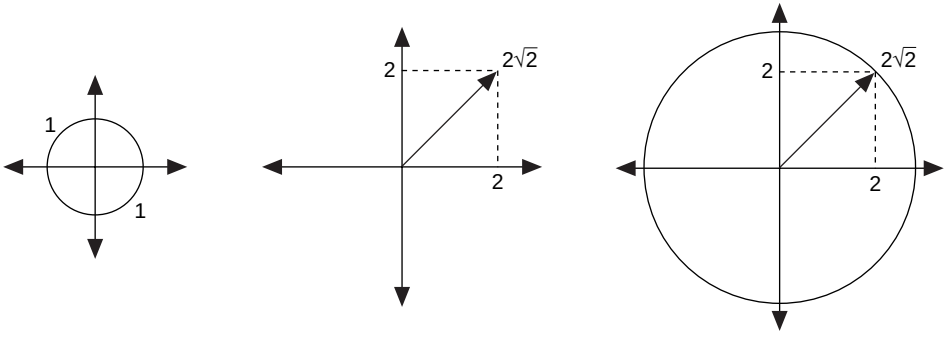


Figure 1a Figure 1b Figure 1c

Figure 1a shows a unit ball of radius = 1 unit. Figure 1b shows a vector of length $\sqrt{2^2 + 2^2} = \sqrt{8} = 2\sqrt{2}$. As shown in Figure 1c, the unit ball must be expanded by a factor of $2\sqrt{2}$ before it can exactly encompass the given vector. Hence, the Euclidean norm of the vector is $2\sqrt{2}$.

The norm of a matrix is defined in terms of an underlying vector norm. It is the maximum relative stretching that the matrix does to any vector. With the vector *2-norm*, the unit ball expands by a factor equal to the norm. On the other hand, with the matrix *2-norm*, the unit ball may become an ellipsoidal (ellipse in 3-D), with some axes longer than others. The longest axis determines the norm of the matrix.

Some matrix norms are much easier to compute than others. The *1-norm* is obtained by finding the sum of the absolute value of all the elements in each column of the matrix. The largest of these sums is called the 1-norm. In mathematical terms, the 1-norm is simply the maximum absolute column sum of the matrix.

$$\|\boldsymbol{A}\|_1 = max_j \sum_{i=0}^{n-1} |a_{i,j}|$$

For example,

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

then

$$\|A\|_1 = max(3, 7) = 7.$$

The *inf-norm* of a matrix is the maximum absolute row sum of the matrix

$$\|A\|_\infty = max_i \sum_{j=0}^{n-1} |a_{i,j}|$$

In this case, you add the magnitudes of all elements in each row of the matrix. The maximum value that you get is called the inf-norm. For the above example matrix,

$$\|A\|_\infty = max(4, 6) = 6.$$

The *2-norm* is the most difficult to compute because it is given by the largest singular value of the matrix. Singular values are discussed in the section *Matrix Factorization*.

Determining Singularity (Condition Number)

Whereas the norm of the matrix provides a way to measure the magnitude of the matrix, the *condition number* of a matrix is a measure of how close the matrix is to being singular. The condition number of a square nonsingular matrix is defined as

$$cond(\mathbf{A}) = \|\mathbf{A}\|_p \cdot \|A^{-1}\|_p$$

where *p* can be one of the four norm types discussed above. For example, to find the condition number of a matrix *A*, you can find the 2-norm of *A*, the 2-norm of the inverse of the matrix *A*, denoted by A^{-1}, and then multiply them together (the inverse of a square matrix A is a square matrix B such that $AB=I$, where I is the identity matrix). As mentioned earlier, the 2-norm

is difficult to calculate on paper. You can use the Matrix Norm VI from the LabVIEW Analysis Library to compute the 2-norm. For example,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, A^{-1} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}, \|A\|_2 = 5.4650, \|A^{-1}\|_2$$

$$= 2.7325, cond(A) = 14.9331$$

The condition number can vary between 1 and infinity. A matrix with a large condition number is nearly singular, while a matrix with a condition number close to 1 is far from being singular. The matrix *A* above is nonsingular. However, consider the matrix

$$B = \begin{bmatrix} 1 & 0.99 \\ 1.99 & 2 \end{bmatrix}.$$

The condition number of this matrix is 47168, and hence the matrix is close to being singular. As you might recall, a matrix is singular if its determinant is equal to zero. However, the determinant is not a good indicator for assessing how close a matrix is to being singular. For the matrix *B* above, the determinant (0.0299) is nonzero; however, the large condition number indicates that the matrix is close to being singular. Remember that the condition number of a matrix is always greater than or equal to one; the latter being true for identity and permutation matrices (a *permutation matrix* is an identity matrix with some rows and columns exchanged). The condition number is a very useful quantity in assessing the accuracy of solutions to linear systems.

In this section, you have become familiar with some basic notation and fundamental matrix concepts such as determinant of a matrix and its rank. The following activity should help you further understand these terms, which will be used frequently throughout the rest of the lesson.

Basic Matrix Operations and Eigenvalues-Eigenvector Problems

In this section, consider some very basic matrix operations. Two matrices, *A* and *B*, are said to be equal if they have the same number of rows and columns and their corresponding elements are all equal. Multiplication of a matrix *A* by a scalar α is equal to multiplication of all its elements by the scalar. That is,

$$\boldsymbol{C} = \alpha\boldsymbol{A} \Rightarrow c_{i,j} = \alpha a_{i,j}$$

For example,

$$2\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Two (or more) matrices can be added or subtracted if and only if they have the same number of rows and columns. If both matrices *A* and *B* have *m* rows and *n* columns, then their sum *C* is an *m*-by-*n* matrix defined as $C = A \pm B$, where $c_{i,j} = a_{i,j} \pm b_{i,j}$. For example,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 2 & 4 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 8 & 5 \end{bmatrix}$$

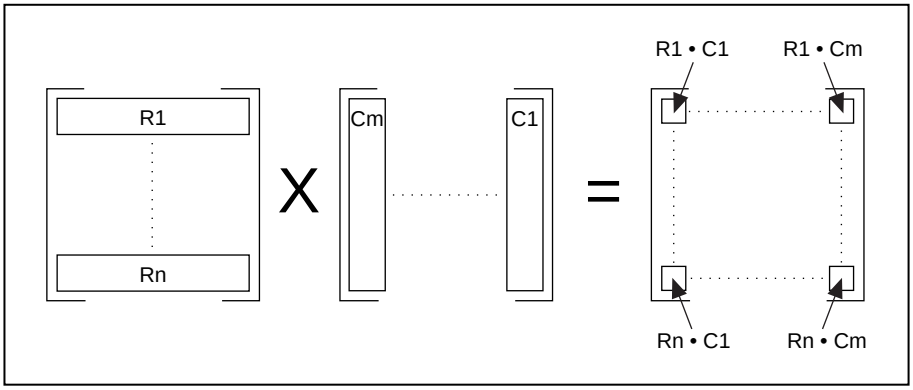
For multiplication of two matrices, the number of columns of the first matrix must be equal to the number of rows of the second matrix. If matrix *A* has *m* rows and *n* columns and matrix *B* has *n* rows and *p* columns, then their product *C* is an *m*-by-*p* matrix defined as $C = AB$, where

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

For example,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 4 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 6 \\ 26 & 16 \end{bmatrix}$$

So, you multiply the elements of the first row of *A* by the corresponding elements of the first column of *B* and add all the results to get the elements in the first row and first column of *C*. Similarly, to calculate the element in the i^{th} row and the j^{th} column of *C*, multiply the elements in the i^{th} row of *A* by the corresponding elements in the j^{th} column of *C*, and then add them all. This is shown pictorially as:



Matrix multiplication, in general, is not commutative, that is, $AB \neq BA$. Also, remember that multiplication of a matrix by an identity matrix results in the original matrix.

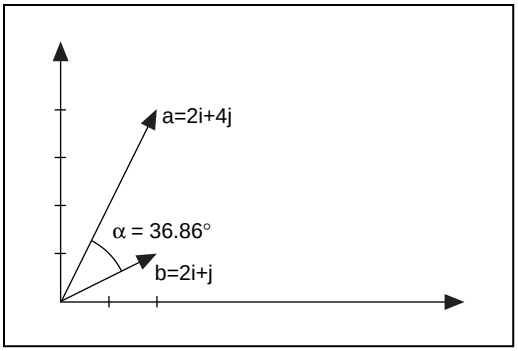
Dot Product and Outer Product

If *X* represents a vector and *Y* represents another vector, then the *dot product* of these two vectors is obtained by multiplying the corresponding elements of each vector and adding the results. This is denoted by

$$X \bullet Y = \sum_{i=0}^{n-1} x_i y_i$$

where *n* is the number of elements in *X* and *Y*. Note that both vectors must have the same number of elements. The dot product is a scalar quantity, and has many practical applications.

For example, consider the vectors $a = 2i + 4j$ and $b = 2i + j$ in a two-dimensional rectangular coordinate system.



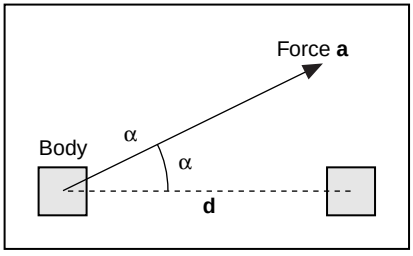
Then the dot product of these two vectors is given by

$$d = \begin{bmatrix} 2 \\ 4 \end{bmatrix} \bullet \begin{bmatrix} 2 \\ 1 \end{bmatrix} = (2 \times 2) + (4 \times 1) = 8$$

The angle α between these two vectors is given by

$$\alpha = inv\cos\left(\frac{a \bullet b}{|a||b|}\right) = inv\cos\left(\frac{8}{10}\right) = 36.86^{o},$$

where |**a**| denotes the magnitude of *a*.



As a second application, consider a body on which a constant force *a* acts. The work W done by *a* in displacing the body is defined as the product of |*d*| and the component of *a* in the direction of displacement *d*. That is,

$$W = |a||d|\cos\alpha = a \bullet d$$

On the other hand, the *outer product* of these two vectors is a matrix. The $(i,j)^{th}$ element of this matrix is obtained using the formula

$$a_{i,j} = x_i \times y_j$$

For example,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \times \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}$$

Eigenvalues and Eigenvectors

To understand eigenvalues and eigenvectors, start with the classical definition. Given an $n \times n$ matrix A, the problem is to find a scalar λ and a nonzero vector x such that

$$Ax = \lambda x$$

Such a scalar λ is called an *eigenvalue*, and x is a corresponding *eigenvector.*

Calculating the eigenvalues and eigenvectors are fundamental principles of linear algebra and allow you to solve many problems such as systems of differential equations when you understand what they represent. Consider an eigenvector x of a matrix A as a nonzero vector that does not rotate when x is multiplied by A (except perhaps to point in precisely the opposite direction). x may change length or reverse its direction, but it will not turn sideways. In other words, there is some scalar constant λ such that the above equation holds true. The value λ is an *eigenvalue* of A.

Consider the following example. One of the eigenvectors of the matrix A, where

$$A = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix},$$

is

$$x = \begin{bmatrix} 0.62 \\ 1.00 \end{bmatrix}.$$

Multiplying the matrix A and the vector x simply causes the vector x to be expanded by a factor of 6.85. Hence, the value 6.85 is one of the eigenvalues of the vector x. For any constant α, the vector αx is also an eigenvector with eigenvalue λ, because

$$A(\alpha x) = \alpha Ax = \lambda \alpha x$$

In other words, an eigenvector of a matrix determines a direction in which the matrix expands or shrinks any vector lying in that direction by a scalar multiple, and the expansion or contraction factor is given by the corresponding eigenvalue. A *generalized* eigenvalue problem is to find a scalar λ and a nonzero vector x such that

$$Ax = \lambda Bx$$

where B is another $n \times n$ matrix.

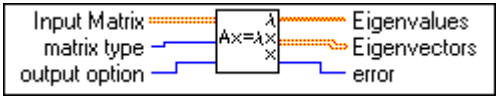
The following are some important properties of eigenvalues and eigenvectors:

- The eigenvalues of a matrix are not necessarily all distinct. In other words, a matrix can have multiple eigenvalues.
- All the eigenvalues of a real matrix need not be real. However, complex eigenvalues of a real matrix must occur in complex conjugate pairs.
- The eigenvalues of a diagonal matrix are its diagonal entries, and the eigenvectors are the corresponding columns of an identity matrix of the same dimension.
- A real symmetric matrix always has real eigenvalues and eigenvectors.
- As discussed earlier, eigenvectors can be scaled arbitrarily.

There are many practical applications in the field of science and engineering for an eigenvalue problem. For example, the stability of a structure and its natural modes and frequencies of vibration are determined by the eigenvalues and eigenvectors of an appropriate matrix. Eigenvalues are also very useful in analyzing numerical methods, such as convergence analysis of iterative methods for solving systems of algebraic equations, and the stability analysis of methods for solving systems of differential equations.

The EigenValues and Vectors VI is shown below. The **Input Matrix** is an N-by-N real square matrix. **Matrix type** determines the type of the input matrix. **Matrix type** could be *0*, indicating a *general matrix*, or 1, indicating a *symmetric matrix*. A symmetric matrix always has real

eigenvalues and eigenvectors. A general matrix has no special property such as symmetry or triangular structure.



Output option determines what needs to be computed. Output option = 0 indicates that only the eigenvalues need to be computed. Output option = 1 indicates that both the eigenvalues and the eigenvectors should be computed. It is computationally very expensive to compute both the eigenvalues and the eigenvectors. So, it is important that you use the output option control in the EigenValues and Vectors VI very carefully. Depending on your particular application, you might just want to compute the eigenvalues or both the eigenvalues and the eigenvectors. Also, a symmetric matrix needs less computation than an nonsymmetric matrix. So, choose the matrix type control carefully.

In this section, you learned about some basic matrix operations and the eigenvalues-eigenvectors problem. The next example will introduce some VIs in the analysis library that perform these operations.

Matrix Inverse and Solving Systems of Linear Equations

The *inverse*, denoted by A^{-1}, of a square matrix A is a square matrix such that

$$A^{-1}A = AA^{-1} = I$$

where I is the identity matrix. The inverse of a matrix exists if and only if the determinant of the matrix is not zero, (that is, it is nonsingular). In general, you can find the inverse of only a square matrix. You can, however, compute the *pseudoinverse* of a rectangular matrix, as discussed later in the *Matrix Factorization* section.

Solutions of Systems of Linear Equations

In matrix-vector notation, a system of linear equations has the form $Ax = b$, where A is a $n \times n$ matrix and b is a given n-vector. The aim is to determine x, the unknown solution n-vector. There are two important questions to be asked about the existence of such a solution. Does such a solution exist, and if it does is it unique? The answer to both of these questions lies in determining the singularity or nonsingularity of the matrix A.

As discussed earlier, a matrix is said to be singular if it has any one of the following equivalent properties:

- The inverse of the matrix does not exist.
- The determinant of the matrix is zero.
- The rows (or columns) of A are linearly dependent.
- $Az = 0$ for some vector $z \neq 0$.

Otherwise, the matrix is nonsingular. If the matrix is nonsingular, its inverse A^{-1} exists, and the system $Ax = b$ has a unique solution: $x = A^{-1}b$ regardless of the value for b. On the other hand, if the matrix is singular, then the number of solutions is determined by the right-hand-side vector b. If A is singular and $Ax = b$, then $A(x + \Upsilon z) = b$ for any scalar Υ, where the vector z is as in the last definition above. Thus, if a singular system has a solution, then the solution cannot be unique.

It is not a good idea to explicitly compute the inverse of a matrix, because such a computation is prone to numerical inaccuracies. Therefore, it is not a good strategy to solve a linear system of equations by multiplying the inverse of the matrix A by the known right-hand-side vector. The general strategy to solve such a system of equations is to transform the original system into one whose solution is the same as that of the original system, but is easier to compute. One way to do so is to use the Gaussian Elimination technique. See Appendix A, *Analysis References*, for more information on matrix computations. The three basic steps involved in the Gaussian Elimination technique are as follows. First, express the matrix A as a product

$$A = LU$$

where L is a unit lower triangular matrix and U is an upper triangular matrix. Such a factorization is known as LU factorization. Given this, the linear system $Ax = b$ can be expressed as $LUx = b$. Such a system can then be solved by first solving the lower triangular system $Ly = b$ for y by

forward-substitution. This is the second step in the Gaussian Elimination technique. For example, if

$$l = \begin{bmatrix} a & 0 \\ b & c \end{bmatrix} \qquad y = \begin{bmatrix} p \\ q \end{bmatrix} \qquad b = \begin{bmatrix} r \\ s \end{bmatrix}$$

then

$$p = \frac{r}{a}, q = \frac{(s - bp)}{c}.$$

The first element of y can be easily determined due to the lower triangular nature of the matrix L. Then you can use this value to compute the remaining elements of the unknown vector sequentially. Hence, the name forward-substitution. The final step involves solving the upper triangular system $Ux = y$ by *back-substitution*. For example, if

$$U = \begin{bmatrix} a & b \\ 0 & c \end{bmatrix} \qquad x = \begin{bmatrix} m \\ n \end{bmatrix} \qquad y = \begin{bmatrix} p \\ q \end{bmatrix}$$

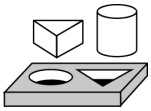
then

$$n = \frac{q}{c}, m = \frac{(p - bn)}{a}.$$

In this case, this last element of **x** can be easily determined and then used to determine the other elements sequentially. Hence, the name back-substitution. So far, this chapter has discussed the case of square matrices. Because a nonsquare matrix is necessarily singular, the system of equations must have either no solution or a nonunique solution. In such a situation, you usually find a unique solution x that satisfies the linear system in an approximate sense.

The Analysis library provides VIs for computing the inverse of a matrix, computing LU decomposition of a matrix, and solving a system of linear equations. It is important to identify the input matrix properly, as it helps avoid unnecessary computations, which in turn helps to minimize numerical inaccuracies. The four possible matrix types are general matrices, positive definite matrices, and lower and upper triangular matrices. A real matrix is positive definite if and only if it is symmetric and the quadratic form for all nonzero vectors is X. If the input matrix is square, but does not have a full rank (a *rank-deficient matrix*), then the VI finds the *least square* solution x. The least square solution is the one which

minimizes the norm of $Ax - b$. The same holds true also for nonsquare matrices.



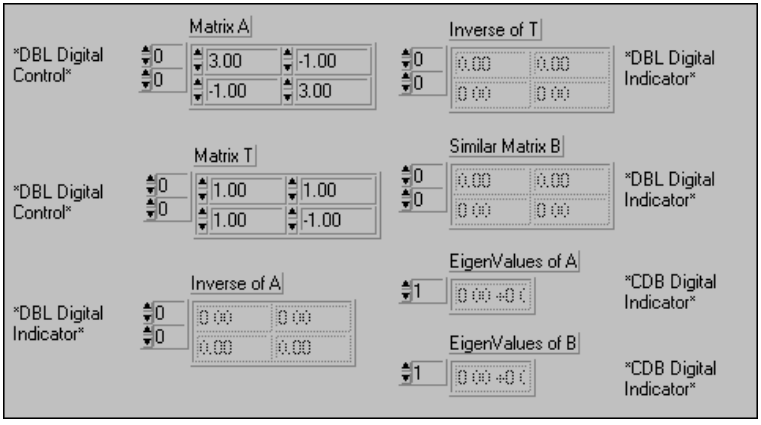
Activity 18-1. Compute the Inverse of a Matrix

Your objective is to compute the inverse of a matrix.

You will build a VI that will compute the inverse of a matrix *A*. Further, you will compute a matrix *B* which is *similar* to matrix *A*. A matrix *B* is similar to a matrix *A* if there is a nonsingular matrix *T* such that $B = T^{-1}AT$ so that *A* and *B* have the same eigenvalues. You will verify this definition of similar matrices.

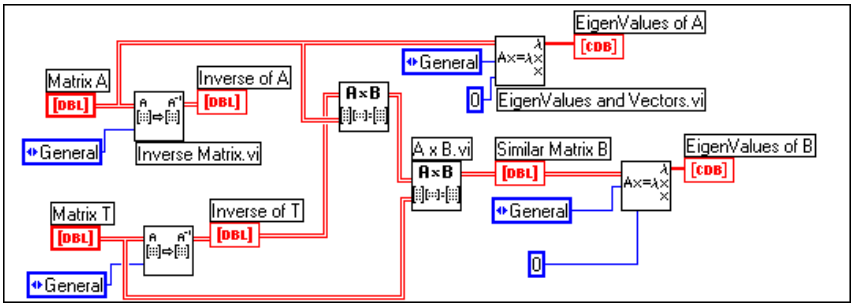
Front Panel

1. Build the front panel as shown below. Matrix *A* is a 2×2 real matrix. Matrix *T* is a 2×2 nonsingular matrix that will be used to construct the similar matrix *B*.



Block Diagram

2. Open the block diagram and modify it as shown in the following illustration.

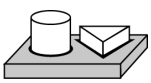


Inverse Matrix function (**Analysis»Linear Algebra** subpalette). In this activity, this function computes the inverse of the input matrix *A*.

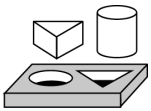
AxB function (**Analysis»Linear Algebra** subpalette). In this activity, this function multiplies two two-dimensional input matrices.

EigenValues and Vectors function (**Analysis»Linear Algebra** subpalette). In this activity, this VI computes the eigenvalues and eigenvectors of the input matrix.

3. Save the VI as `Matrix Inverse.vi` in the `LabVIEW\Activity` directory.
4. Return to the front panel and run the VI. Check if the eigenvalues of *A* and the similar matrix *B* are the same.



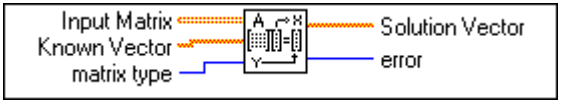
End of Activity 18-1.



Activity 18-2. Solve a System of Linear Equations

Your objective is to solve a system of linear equations.

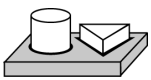
Many practical applications require you to solve a system of linear equations. A very important area of application is related to military defense. This includes analysis of electromagnetic scattering and radiation from large targets, performance analysis of large radomes, and design of aerospace vehicles having low radar cross sections (the Stealth Technology). A second area of application is in the design and modeling of wireless communication systems such as hand-held cellular phones. This list of applications goes on and on, and therefore it is very important for you to properly understand how to use the VIs in the Analysis Library to solve a linear system of equations.



1. Use the `Solve Linear Equations.vi` in the **Analysis»Linear Algebra** subpalette to solve the system of equations $Ax = b$ where the Input Matrix A and the Known Vector b are

$$A = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -1 & 7 \end{bmatrix}, b = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}$$

   Choose matrix type equal to general.
2. Use the A x Vector.vi to multiply the matrix A and the vector x (output of the above operation) and check if the result is equal to the vector b above.
3. Save the VI as `Linear System.vi` in the `LabVIEW\Activity` directory.



End of Activity 18-2.

Matrix Factorization

The previous section discussed how a linear system of equations can be transformed into a system whose solution is simpler to compute. The basic idea was to factorize the input matrix into the multiplication of several, simpler matrices. You looked at one such technique, the *LU decomposition* technique, in which you factorized the input matrix as a product of upper and lower triangular matrices. Other commonly used factorization methods are *Cholesky*, *QR*, and the *Singular Value Decomposition (SVD)*. You can use these factorization methods to solve many matrix problems, such as solving linear system of equations, inverting a matrix, and finding the determinant of a matrix.

If the input matrix *A* is symmetric and positive definite, then an LU factorization can be computed such that $A = U^T U$, where *U* is an upper triangular matrix. This is called *Cholesky factorization*. This method requires only about half the work and half the storage compared to LU factorization of a general matrix by Gaussian elimination. It is easy to determine if a matrix is positive definite by using the Test Positive Definite VI in the Analysis library.

A matrix *Q* is *orthogonal* if its columns are *orthonormal*. That is, if $Q^T Q = I$, the identity matrix. *QR factorization* technique factors a matrix as the product of an orthogonal matrix *Q* and an upper triangular matrix *R*. That is, $A = QR$.QR factorization is useful for both square and rectangular matrices. A number of algorithms are possible for QR factorization, such as the *Householder transformation*, the *Givens transformation* and the *Fast Givens Transformation*.

The Singular Value Decomposition (SVD) method decomposes a matrix into the product of three matrices: $A = USV^T$. *U* and *V* are orthogonal matrices. *S* is a diagonal matrix whose diagonal values are called the *singular values* of *A*. The singular values of *A* are the nonnegative square roots of the eigenvalues of $A^T A$, and the columns of *U* and *V*, which are called left and right singular vectors, are orthonormal eigenvectors of AA^T and $A^T A$, respectively. SVD is useful for solving analysis problems such as computing the rank, norm, condition number, and pseudoinverse of matrices. The following section discusses this last application.

Pseudoinverse

The pseudoinverse of a scalar σ is defined as $1/\sigma$ if $\sigma \neq 0$, and zero otherwise. In case of scalars, pseudoinverse is the same as the inverse. You can now define the pseudoinverse of a diagonal matrix by transposing the matrix and then taking the scalar pseudoinverse of each entry. Then the pseudoinverse of a general real $m \times n$ matrix A, denoted by $A^{\dagger}$, is given by

$$A^{\dagger} = VS^{\dagger}U^{T}$$

Note that the pseudoinverse exists regardless of whether the matrix is square or rectangular. If A is square and nonsingular, then the pseudoinverse is the same as the usual matrix inverse. The Analysis Library includes a VI for computing the pseudoinverse of real and complex matrices.

Summary

- A matrix can be considered as a two-dimensional array of *m* rows and *n* columns. Determinant, rank, and condition number are some important attributes of a matrix.
- The condition number of a matrix affects the accuracy of the final solution.
- The determinant of a diagonal matrix, an upper triangular matrix, or a lower triangular matrix is the product of its diagonal elements.
- Two matrices can be multiplied only if the number of columns of the first matrix is equal to the number of rows in the second matrix.
- An eigenvector of a matrix is a nonzero vector that does not rotate when the matrix is applied to it. Similar matrices have the same eigenvalues.
- The existence of a unique solution for a system of equations depends on whether the matrix is singular or nonsingular.

19

Probability and Statistics

This chapter explains some fundamental concepts on probability and statistics and shows how to use these concepts in solving real-world problems. For examples of how to use the probability and statistics VIs, see the examples located in `examples\analysis\statxmpl.llb`.

Probability and Statistics

We live in an information age in which facts and figures form an important part of life. Statements such as “There is a 60% chance of thunderstorms,” “Joe was ranked among the top five in the class,” “Michael Jordan has an average of 30 points a game this season,” and so on are common. These statements give a lot of information, but we seldom think how this information was obtained. Was there a lot of data involved in obtaining this information? If there was, how did someone condense it to single numbers such as *60% chance* and *average of 30 points* or terms such as *top five*. The answer to all these questions brings up the very interesting field of statistics.

First, consider how information (data) is generated. Consider the 1997 basketball season. Michael Jordan of the Chicago Bulls played 51 games, scoring a total of 1568 points. This includes the 45 points he posted, including the game-winning buzzer three-pointer, in a 103-100 victory over the Charlotte Hornets; his 36 points in an 88-84 victory over the Portland Trail Blazers; a season high of 51 points in an 88-87 victory over the New York Nicks; 45 points, 7 rebounds, 5 assists and 3 steals in a 102-97 victory over the Cleveland Cavaliers; and his 40 points, 6 rebounds, and 6 assists in a 107-104 victory over the Milwaukee Bucks. The point is not that Jordan is a great player, but that a single player can generate lots of data in a single season. The question is how to condense all this data so that it brings out all the essential information and is yet easy to remember. This is where the term *statistics* comes into the picture.

To condense all the data, single numbers must make it more intelligible and help draw useful inferences. For example, consider the number of points that Jordan scored in different games. It is difficult to remember how many points he scored in each game. But if you divide the total number of points

that Jordan scored (1568) by the number of games he has played (51), you have a single number of 30.7 and can call it points per game *average*.

Suppose you want to rate Jordan's free throw shooting skills. It might be difficult to do so by looking at his performance in each game. However, you can divide the number of free throws he has scored in all the games by the total number of free throws he was awarded. This shows he has a free throw *percentage* of 84.4%. You can obtain this number for all the NBA players and then rank them. Thus, you can condense the information for all the players into single numbers representing free throw percentage, points per game, and three-point average. Based on this information, you can rank players in different categories. You can further weight these different numbers and come up with a single number for each player. These single numbers can then help us in judging the Most Valuable Player (MVP) for the season. Thus, in a broad sense, the term statistics implies different ways to summarize data to derive useful and important information from it.

The next question is, what is probability? You have looked at ways to summarize lots of data into single numbers. These numbers then help draw conclusions for the present. For example, looking at Jordan's statistics for the 1996 season helped elect him the MVP for that season. But can you say anything about the future? Can you measure the degree of accuracy in the inference and use it for making future decisions? The answer lies in the theory of probability. Whereas, in laymen's terms, one would say that it is *probable* that Jordan will continue to be the best in the years to come, you can use different concepts in the field of probability, as discussed later in this chapter, to make more quantitative statements.

In a completely different scenario, there may be certain experiments whose outcomes cannot be predetermined, but certain outcomes may be more probable. This once again leads to the notion of probability. For example, if you flip an unbiased coin in the air, what is the chance that it will land heads up? The chance or probability is 50%. That means, if you repeatedly flip the coin, half the time it will land heads up. Does this mean that 10 tosses will result in exactly five heads? Will 100 tosses result in exactly 50 heads? Probably not. But in the long run, the probability will work out to be 0.5.

To summarize, whereas statistics allows you to summarize data and draw conclusions for the present, probability allows you to measure the degree of accuracy in those conclusions and use them for the future.

Statistics

In this section, you will look at different concepts and terms commonly used in statistics and see how to use the G Analysis VIs in different applications.

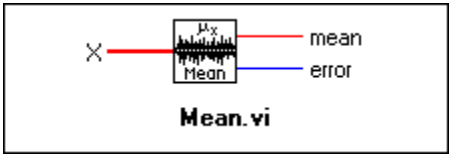
Mean

Consider a data set *X* consisting of *n* samples $x_0, x_1, x_2, x_3, \ldots, x_{n-1}$. The mean value (a.k.a. average) is denoted by $\bar{x}$ and is defined by the formula

$$\bar{x} = \frac{1}{n}(x_0 + x_1 + x_2 + x_3 + \ldots + x_{n-1})$$

In other words, it is the sum of all the sample values divided by the number of samples. As in the Michael Jordan example, the data set consisted of 51 samples. Each sample was equal to the number of points that Jordan scored in each game. The total of all these points was 1568, divided by the number of samples (51) to get a mean or average value of 30.7.

The input-output connections for the Mean VI are shown below.



Median

Let $S = \{s_0, s_1, s_2, \ldots, s_{n-1}\}$ represent the sorted sequence of the data set *X*. The sequence can be sorted either in the ascending order or in descending order. The median of the sequence is denoted by x_{median} and is obtained by the formula

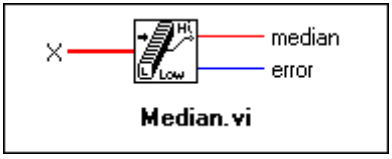
$$x_{median} = \begin{cases} s_i & \text{n is odd} \\ 0.5(s_{k-1} + s_k) & \text{n is even} \end{cases}$$

where

$$i = \frac{n-1}{2} \text{ and } k = \frac{n}{2}.$$

In words, the median of a data sequence is the *midpoint* value in the sorted version of that sequence. For example, consider the sequence $\{5, 4, 3, 2, 1\}$ consisting of five (odd number) samples. This sequence is already sorted in the descending order. In this case, the median is the midpoint value, 3. Consider a different sequence $\{1, 2, 3, 4\}$ consisting of four (even number) samples. This sequence is already sorted in the ascending order. In this case, there are two midpoint values, 2 and 3. As per the formula above, the median is equal to $0.5 \times (2 + 3) = 2.5$. If a student X scored 4.5 points on a test and another student Y scored 1 point on the same test, the median is a very useful quantity for making qualitative statements such as "X lies in the top half of the class" or "Y lies in the bottom half of the class."

The input-output connections for the Median VI are shown below.



Sample Variance

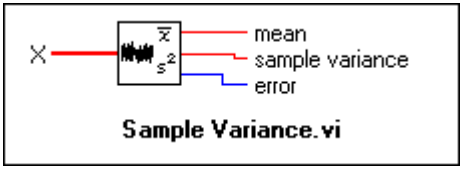
The sample variance of the data set *X* consisting of *n* samples is denoted by s^2 and is defined by the formula

$$s^2 = \frac{1}{n-1}[(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \ldots + (x_n - \bar{x})^2]$$

where $\bar{x}$ denotes the mean of the data set. Hence, the sample variance is equal to the sum of the squares of the deviations of the sample values from the mean divided by *n–1*.

Note ***The above formula does not apply for n=1. However, it does not mean anything to compute the sample variance if there is only one sample in the data set.***

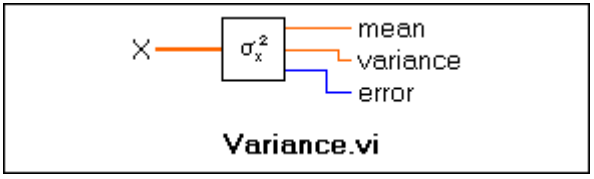
The input-output connections for the Sample Variance VI are shown below.



In other words, the sample variance measures the spread or dispersion of the sample values. If the data set consists of the scores of a player from different games, the sample variance can be used as a measure of the consistency of the player. It is always positive, except when all the sample values are equal to each other and in turn equal to the mean.

There is one more type of variance called population variance. The formula to compute population variance is similar to the one above to compute sample variance, except for the (*n–1*) in the denominator replaced by *n*.

The input-output connections for the Variance VI are shown below



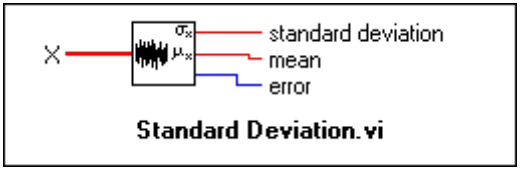
The Sample Variance VI computes sample variance, whereas the Variance VI computes the population variance. Statisticians and mathematicians prefer to use the latter, engineers the former. It really does not matter for large values of n, say $n \geq 30$.

Note ***Use the proper type of VI suited for your application.***

Standard Deviation

The positive square root of the sample variance s^2 is denoted by s and is called the standard deviation of the sample.

The input-output connections for the Standard Deviation VI are shown below.



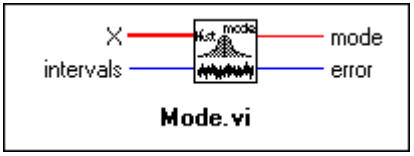
Mode

The mode of a sample is a sample value that occurs most frequently in the sample. For example, if the input sequence X is

$$X = \{0, 1, 3, 3, 4, 4, 4, 5, 5, 7\}$$

then the mode of X is 4, because that is the value that most often occurs in X.

The input-output connections for the Mode VI are shown below.



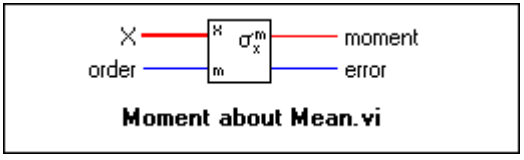
Moment About Mean

If X represents the input sequence with n number of elements in it, and $\bar{x}$ is the mean of this sequence, then the m^{th}-order moment can be calculated using the formula

$$\sigma_x^{\,m} = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \bar{x})^m$$

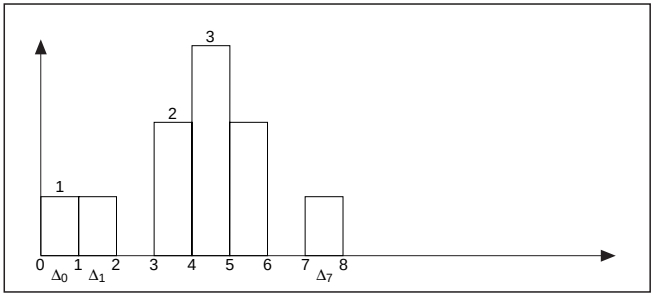
In other words, the moment about mean is a measure of the deviation of the elements in the sequence from the mean. Note that for $m = 2$, the moment about mean is equal to the population variance.

The input-output connections for the Moment About Mean VI are shown below.



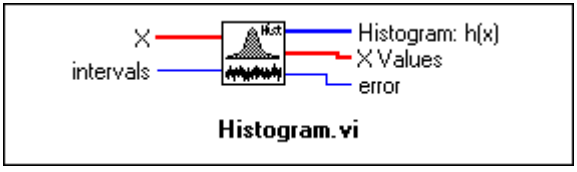
Histogram

So far, this chapter has discussed different ways to extract important features of a data set. The data is usually stored in a table format, which many people find difficult to grasp. The visual display of data helps us gain insights into the data. Histogram is one such graphical method for displaying data and summarizing key information. Consider a data sequence $X = \{0, 1, 3, 3, 4, 4, 4, 5, 5, 8\}$. Divide the total range of values into 8 intervals. These intervals are 0 – 1, 1 – 2, 2 – 3, ..., 7 – 8. The histogram for the sequence *X* then plots the number of data samples that lie in that interval, not including the upper boundary.



The figure above shows that one data sample lies in the range 0 – 1 and 1 – 2, respectively. However, there is no sample in the interval 2 – 3. Similarly, two samples lie in the interval 3 – 4, and three samples lie in the range 4 – 5. Examine the data sequence *X* above and be sure you understand this concept.

There are different ways to compute data for histogram. Next you will see how it is done in the Histogram VI using the sequence *X*.



As shown above, the inputs to this VI are the input sequence *X* and the number of intervals *m*. The VI obtains **Histogram:h(x)** as follows. It scans

X to determine the range of values in it. Then the VI establishes the interval width, Δx, according to the specified value of *m*

$$\Delta x = \frac{max - min}{m}$$

where *max* is the maximum value found in *X*, min is the minimum value found in *X*, and *m* is the specified number of intervals.

Let

$$m = 8.$$

Then

$$\Delta x = \frac{8 - 0}{8} = 1$$

Let χ represent the output sequence *X* Values. The histogram is a function of *X*. This VI evaluates the elements of χ using

$$\chi_i = min + 0.5\Delta x + i\Delta x \qquad for \qquad i = 0, 1, 2, \ldots, m - 1$$

For this example,

$$\chi_0 = 0.5, \chi_1 = 1.5, \ldots, \chi_7 = 7.5.$$

The VI then defines the i^{th} interval to be in the range of values from $\chi_i - 0.5\Delta x$ up to but not including $\chi_i + 0.5\Delta x$,

$$\Delta_i = [(\chi_i - 0.5\Delta x), (\chi_i + 0.5\Delta x)], for \qquad i = 0, 1, 2, \ldots, m - 1$$

and defines the function $y_i(x) = 1$ for *x* belonging to Δ_i and zero elsewhere. The function has unity value if the value of *x* falls within the specified interval, not including the boundary. Otherwise, it is zero. Notice that the interval is centered about χ_i and its width is Δ_x. If a value is equal to max, it is counted as belonging to the last interval.

For our example,

$$\Delta_0 = [0, 1], \Delta_1 = [1, 2], \ldots, \Delta_7 = [7, 8]$$

and as an example

$$y_0(0) = 1$$

and

$$y_0(1) = y_0(3) = y_0(4) = y_0(5) = y_0(8) = 0.$$

Finally, the VI evaluates the histogram sequence *H* using

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \qquad for \qquad i = 0, 1, 2, \ldots, m-1$$

where h_i represents the elements of the output sequence *Histogram: h(X)* and *n* is the number of elements in the input sequence *X*. For this example, $h_0 = 1, h_4 = 3, \ldots, h_7 = 1$.

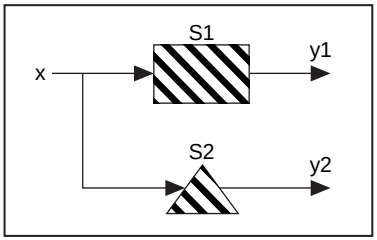
The G Analysis Library also has a General Histogram VI that is more advanced than the Histogram VI. Please refer to the *Analysis Online Reference* for detailed information.

Mean Square Error (MSE)

If X and Y represent two input sequences, then the mean square error is the average of the sum of the square of the difference between the corresponding elements of the two input sequences. The following formula is used to find the mse.

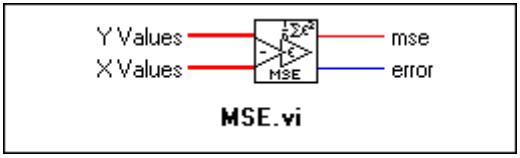
$$mse = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - y_i)^2$$

where *n* is the number of data points.



Consider a digital signal x fed to a system, S1. The output of this system is y1. Now you acquire a new system, S2, which is theoretically known to generate the same result as S1 but has two times faster response time. Before replacing the old system, you want to be absolutely sure that the output response of both the systems is the same. If the sequences y1 and y2 are very large, it is difficult to compare each element in the sequences. In such a scenario, you can use the MSE VI to calculate the mean square error (mse) of the two sequences y1 and y2. If the mse is smaller than an acceptable tolerance, then the system S1 can be reliably replaced by the new system S2.

The input-output connections for the MSE VI are shown below.



Root Mean Square (RMS)

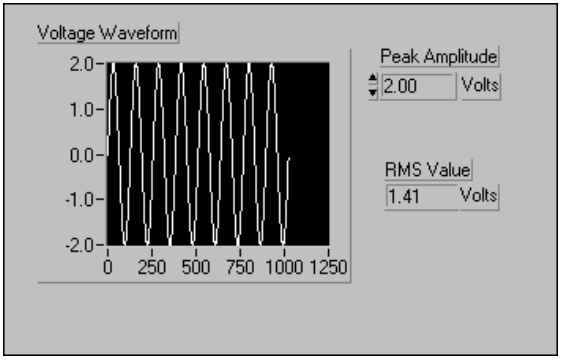
The root mean square Ψ_x of a sequence X is the positive square root of the mean of the square of the input sequence. In other words, you can square the input sequence, take the mean of this new squared sequence, and then take the square root of this quantity. The formula used to compute the rms value is

$$\Psi_x = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^2}$$

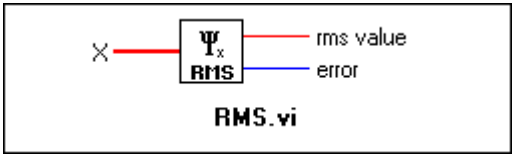
where n is the number of elements in X.

RMS is a widely used quantity in the case of analog signals. For a sine voltage waveform, if V_p is the peak amplitude of the signal, then the root mean square voltage V_{rms} is given by $\frac{V_p}{\sqrt{2}}$.

The following figure shows a voltage waveform of peak amplitude = 2 V and the RMS value of $\sqrt{2} \approx 1.41$ V computed using the Analysis Library.



The input-output connections for the RMS VI are shown below.



Probability

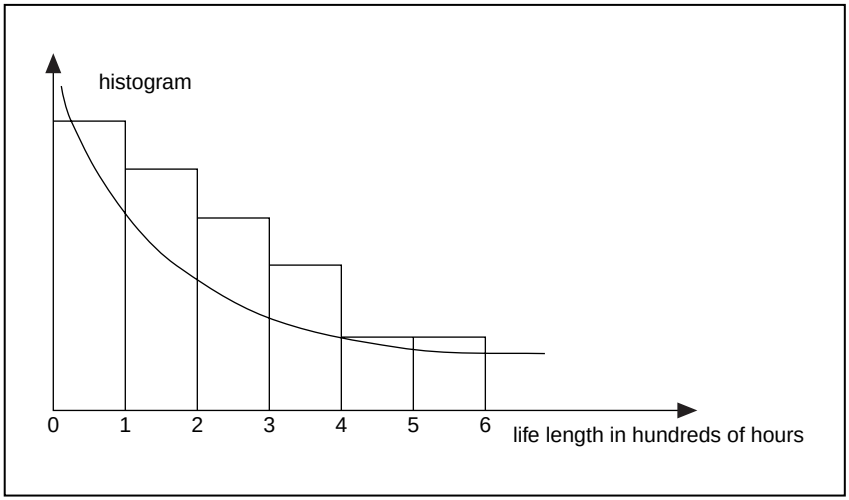
In any random experiment, there is always a chance that a particular event will or will not occur. A number between 0 and 1 is assigned to measure this chance, or probability, that a particular event occurs. If you are absolutely sure that the event will occur, its probability is 100% or 1.0, but if you are sure that the event will not occur, its probability is 0.

Consider a simple example. If you roll a single unbiased die, there are six possible events that can occur—either a 1, 2, 3, 4, 5, or 6 can result. What is the probability that a 2 will result? This probability is one in six, or 0.16666. You can define probability in simple terms as: The probability that an event A will occur is the ratio of the number of outcomes favorable to A to the total number of equally likely outcomes.

Random Variables

Many experiments generate outcomes that you can interpret in terms of real numbers. Some examples are the number of cars passing a stop sign during a day, number of voters favoring candidate A, and number of accidents at a particular intersection. The values of the numerical outcomes of this experiment can change from experiment to experiment and are called random variables. Random variables can be discrete (if they can take on only a finite number of possible values) or continuous. As an example of the latter, weights of patients coming into a clinic may be anywhere from, say, 80 to 300 pounds. Such random variables can take on any value in an interval of real numbers. Given such a situation, suppose you want to find the probability of encountering a patient weighing exactly 172.39 pounds. You will see how to calculate this probability next using an example.

Consider an experiment to measure the life lengths x of 50 batteries of a certain type. These batteries are selected from a larger population of such batteries. The histogram for observed data is shown below.



This figure shows that most of the life lengths are between zero and 100 hours, and the histogram values drop off smoothly you we look at larger life lengths.

You can approximate the histogram shown above by an exponentially decaying curve. You could take this function as a mathematical model for the behavior of the data sample. If you want to know the probability that a randomly selected battery will last longer than four hundred hours, this value can be approximated by the area under the curve to the right of the value 4. Such a function that models the histogram of the random variable is called the *probability density function.*

To summarize all the information above in terms of a definition, a random variable *X* is said to be *continuous* if it can take on the infinite number of

possible values associated with intervals of real numbers, and there is a function *f(x)*, called the *probability density function*, such that

1. $f(x) \geq 0$ for all x

2. $\int_{-\infty}^{\infty} f(x)dx = 1$

3. $P(a \leq X \leq b) = \int_{a}^{b} f(x)dx$

Notice from equation (3) above, that for a specific value of the continuous random variable, that is for

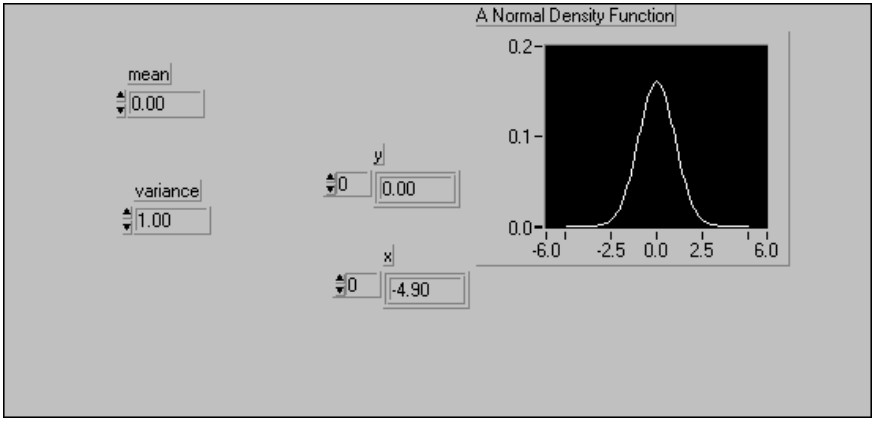
$$X=a,\ P(X = a) = \int_{a}^{a} f(x)dx = 0 .$$

It should not be surprising that you assign a probability of zero to any specific value, because there are an infinite number of possible values that the random variable can take. Therefore, the chance that it will take on a specific value $X = a$ is extremely small.

In the previous example used the exponential function model for the probability density function. There are a number of different choices for this function. One of these is the Normal Distribution, discussed below.

Normal Distribution

The normal distribution is one of the most widely used continuous probability distributions. This distribution function has a symmetric bell shape, as shown in the following illustration.



The curve is centered at the mean value $\bar{x} = 0$, and its spread is measured by the variance $s^2 = 1$. These two parameters completely determine the shape and location of the normal density function, whose functional form is given by

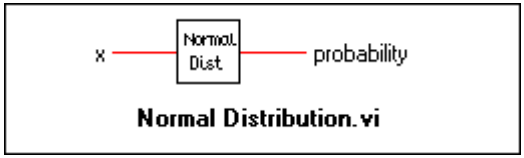
$$f(x) = \frac{1}{\sqrt{2\pi}s} e^{-(x-\bar{x})^2/(2s^2)}$$

Suppose a random variable Z has a normal distribution with mean equal to zero and variance equal to one. This random variable is said to have *standard normal distribution*.

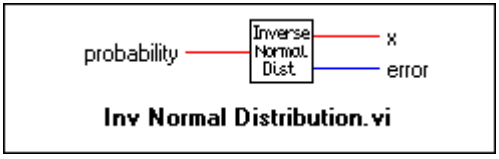
The G Analysis Normal Distribution VI computes the one-sided probability, *p*, of the normally distributed random variable *x*.

$$p = Prob(X \leq x)$$

where *X* is a standard normal distribution with the mean value equal to zero and variance equal to one, *p* is the probability and *x* is the value.

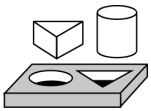


Suppose you conduct an experiment in which you measure the heights of adult males. You conduct this experiment on 1000 randomly chosen men and obtain a data set S. The histogram distribution has many measurements clumped closely about a mean height, with relatively few very short and very tall males in the population. Therefore, the histogram can be closely approximated by a normal distribution. Now suppose that, among a different set of 1000 randomly chosen males, you want to find the probability that the height of a male is greater than or equal to 170 cms. You can use the Normal Distribution VI to find this probability. Set the input $x = 170$. Thus, the choice of the probability density function is fundamental to obtaining a correct probability value.



The Inverse Normal Distribution VI performs exactly the opposite function. Given a probability *p*, it finds the values *x* that have the chance of lying in a normally distributed sample. For example, you might want to find the heights that have a 60% chance of lying in a randomly chosen data set.

As mentioned earlier, there are different choices for the probability density function. The well-known and widely-used ones are the Chi-Square distribution, the F distribution, and the T-distribution. For more information on these distributions, refer to Appendix A, *Analysis References*. The G Analysis library has VIs that computes the one-sided probability for these different types of distributions. In addition, it also has VIs that perform the inverse operation.



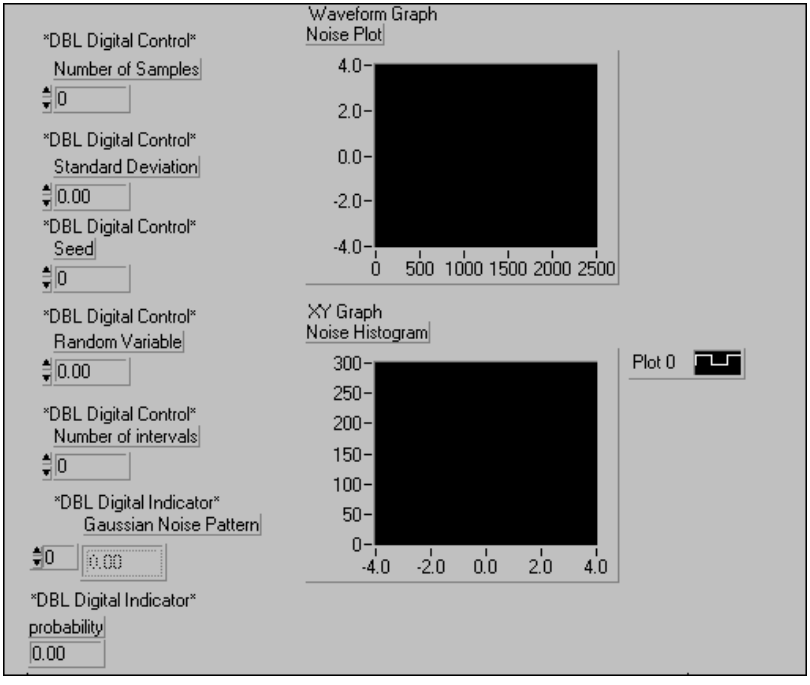
Activity 19-1. Use the Normal Distribution VI

Your objective is to understand key probability concepts.

In this activity, you will first generate a data sample with standard normal distribution and then use the Normal Distribution VI to check the probability of a random variable *x*.

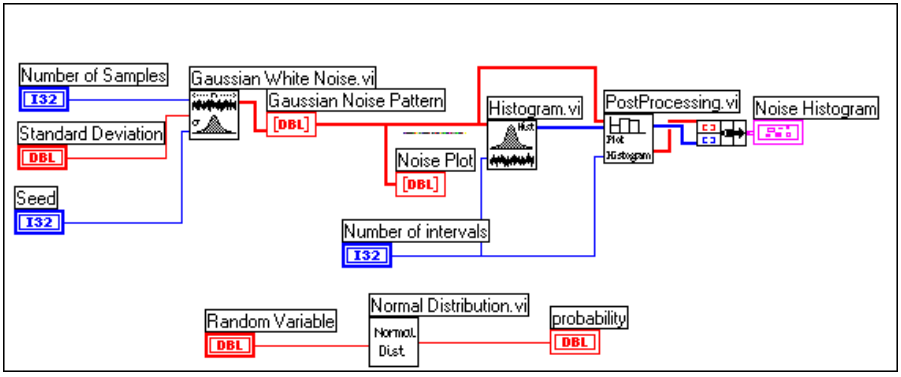
Front Panel

1. Build the front panel as shown in the following figure. NoisePlot is a waveform graph, whereas NoiseHistogram is an XY graph.



Block Diagram

2. Build the block diagram as shown in the following illustration. The Gaussian White Noise generates a Gaussian-distributed pattern with mean value equal to 0 and standard deviation set by the user using the input standard deviation. Samples is the number of samples of the Gaussian Noise Pattern. Seed is the seed value used to generate the random noise. Connect the Gaussian Noise Pattern to the waveform graph Noise Plot.



Gaussian White Noise function (**Analysis»Signal Generation** subpalette). In this activity, this function generates a Gaussian White Noise pattern.

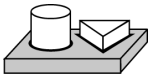
Histogram function (**Analysis»Probability and Statistics** subpalette). In this activity, this function computes the histogram of the Gaussian Noise Pattern.

Normal Distribution function (**Analysis»Probability and Statistics** subpalette). In this activity, this function computes the one-sided probability of the normally distributed random variable **Random Variable**.

3. You will compute the histogram of the Gaussian Noise Pattern using the Histogram VI used in the previous activity.
4. As discussed earlier, do some postprocessing to plot the histogram in a different way. Select the PostProcessing VI from the `LabVIEW\Activity` directory.
5. Bundle the output of this VI and connect it to the Noise Histogram.
6. Select the Normal Distribution VI. Connect the Random Variable control to the input terminal and connect the output to the probability indicator.
7. Return to the front panel. Set the Number of Samples to 2,048, Standard Deviation to 1, Seed to 2 and Number of intervals to 10. Run the VI.
8. You will see the Gaussian white noise on the Noise Plot graph. It is difficult to tell much from this plot. However, the histogram plot for the same noise pattern provides a lot of information. It shows that most of the samples are centered around the mean value of zero. From this histogram, you can approximate this noise pattern by a Normal Distribution function (Gaussian distribution). Because the mean value is zero and you set the standard deviation equal to one, the probability density function is actually a standard normal distribution.

Note ***It is very important that you carefully choose the proper type of distribution function to approximate your data. In this example, you actually plotted the histogram to make this decision. Many times, you can make an intelligent decision based solely on prior knowledge of the behavior and characteristics of the data sample.***

9. Return to the front panel and enter a value for Random Variable. This VI will compute the one-sided probability of this normally distributed random variable. Remember, you have assumed that the variable is normally distributed by looking at the histogram.
10. Save the VI as `Probability.vi` in the `LabVIEW\Activity` directory.



End of Activity 19-1.

Summary

- Different concepts in statistics and probability help decipher information and data to make intelligent decisions.
- Mean, Median, Sample Variance, and Mode are some of the statistics techniques to help in making inferences from a sample to a population.
- Histograms are widely used as a simple but informative method of data display.
- Using the theory of probability, you can make inferences from a sample to a population and then measure the degree of accuracy in those inferences.

Part IV

Network and Interapplication Communication

This section contains basic information about network and interapplication communication.

Part IV, *Network and Interapplication Communication*, contains the following chapters.

- Chapter 20, *Introduction to Communication*, introduces the way LabVIEW handles networking and interapplication communication.
- Chapter 21, *TCP and UDP*, describes Internet Protocol (IP), User Datagram Protocol (UPD), Transmission Control Protocol (TCP), internet addresses, and examples of TCP client/server applications.
- Chapter 22, *ActiveX Support*, explains how to use LabVIEW as an ActiveX server and client. ActiveX is the same as OLE Automation communication.
- Chapter 23, *Using DDE*, describes the LabVIEW VIs for Dynamic Data Exchange (DDE) for Windows 3.1, Windows 95, and Windows NT. These VIs execute DDE functions for sharing data with other applications that accept DDE connections.
- Chapter 24, *AppleEvents*, describes AppleEvents, one form of Macintosh-only interapplication communication (IAC) through which Macintosh applications can communicate.
- Chapter 25, *Program-to-Program Communication*, describes program-to-program communication (PPC), a low-level form of Apple interapplication communication (IAC) by which Macintosh applications send and receive blocks of data.

20

Introduction to Communication

This chapter introduces the way LabVIEW handles networking and interapplication communications.

LabVIEW Communication Overview

For the purpose of this discussion, *networking* refers to communication between multiple processes, possibly on separate computers. This communication usually occurs over a hardware network, such as ethernet or LocalTalk.

A primary use for networking in software applications is for one or more applications to use the services of another application. For example, the application providing services (the server) could be either a data collection application running on a dedicated computer or a database program providing information to other applications.

In this discussion, you are introduced to networking and communication terminology and programming networked applications.

Introduction to Communication Protocols

For communication between processes to occur, the processes must use a common communications language, referred to as a *protocol*.

A communication protocol lets you specify the data that you want to send or receive and the location of the destination or source, without having to worry about how the data gets there. The protocol translates your commands into data that network drivers can accept. The network drivers then take care of transferring data across the network as appropriate.

Several networking protocols have emerged as accepted standards for communications. In general, one protocol is not compatible with another. Thus, in communication applications, one of the first things you must do is decide which protocol to use. If you want to communicate with an existing, off-the-shelf application, then you have to work within the protocols supported by that application.

When you actually write the application, you have more flexibility in choosing a protocol. Factors that affect your protocol choice include the type of machines the processes will run on, the kind of hardware network you have available, and the complexity of the communication that your application will need.

Several protocols are built into LabVIEW, some of which are specific to a type of computer. LabVIEW uses the following protocols to communicate between computers:

- **TCP**—Available on all computers
- **UDP**—Available on all computers
- **DDE**—Available on the PC, for communication between Windows applications
- **ActiveX**—Available for use with Windows 95 and Windows NT
- **AppleEvents**—Available on the Macintosh, for sending messages between Macintosh applications
- **PPC**—Available on the Macintosh, for sending and receiving data between Macintosh applications

Each protocol is different, especially in the way they refer to the network location of a remote application. They are incompatible with each other, so if you want to communicate between a Macintosh and a PC, you must use a protocol compatible with both, such as TCP.

Other LabVIEW communication options include the following:

- **System Exec VI**—Allows you to execute a system level command. There are actually two System Exec VIs, one for use with all versions of Windows, the other with Sun and HP-UX
- **Named Pipes**—Available on UNIX only
- **HiQ**—Available on the Macintosh and PC only

File Sharing Versus Communication Protocols

Before you get too deeply involved in communication protocols, consider whether another approach is more appropriate for your application. For instance, consider an application where a dedicated system acquires data and you want the data recorded on a different computer.

You could write an application that uses networking protocols to send data from the acquisition computer to the data-recording machine, where a separate application collects the data and stores it on disk.

A simpler method might involve the filesharing capabilities available on most networked computers. With filesharing, drivers that are part of the operating system let you connect to other machines. The remote machine's disk storage is treated as an extension of your own disk storage. Once you connect two systems, filesharing usually makes this connection transparent, so that any application can write to the remote disk as if connected locally. Filesharing is frequently the simplest method for transferring data between machines.

Client/Server Model

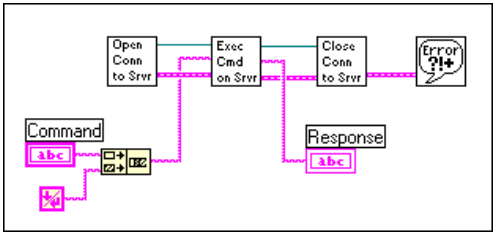
The client/server model is a common model for networked applications. In the client/server model, one set of processes (clients) requests services from another set of processes (servers).

For example, in your application you could set up a dedicated computer for acquiring measurements from the real world. The computer acts as a server when it provides data to other computers on request. It acts as a client when it requests another application, such as a database program, to record the data that it acquires.

In LabVIEW, you can use client and server applications with all protocols except Macintosh AppleEvents. You can use AppleEvents to send commands to other applications. You cannot set up a command server in LabVIEW using AppleEvents. If you need server capabilities on the Macintosh, use either TCP, UDP, or PPC.

A General Model for a Client

The following block diagram shows what a simplified model for a client looks like in LabVIEW.



In the preceding diagram, LabVIEW first opens a connection to a server. It then sends a command to the server, gets a response back, and closes the

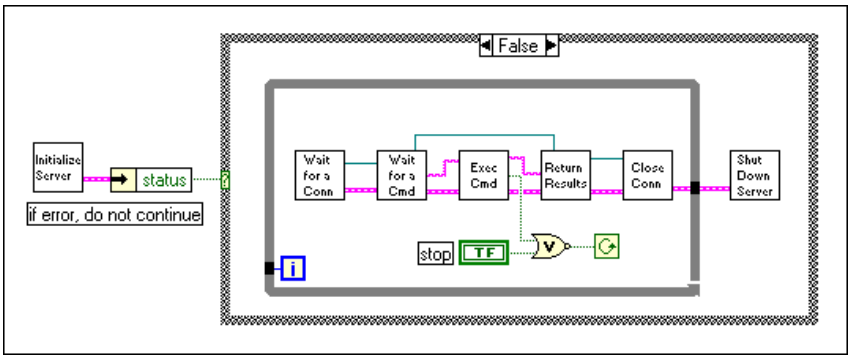
connection to the server. Finally, it reports any errors that occurred during the communication process.

For higher performance, you can process multiple commands once the connection is open. After the commands are executed, you can close the connection.

This basic block diagram structure serves as a model and is used elsewhere in this manual to demonstrate how to implement a given protocol in LabVIEW.

A General Model for a Server

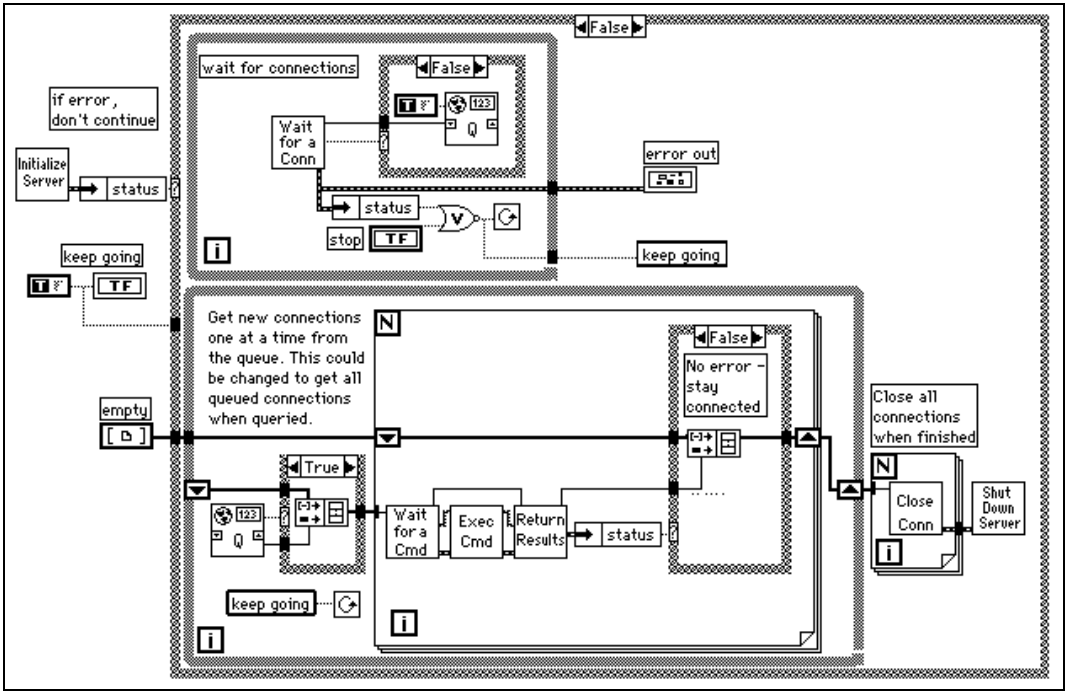
The following block diagram shows a simplified model for a server in LabVIEW.



In the preceding diagram, LabVIEW first initializes the server. If the initialization is successful, LabVIEW goes into a loop, where it waits for a connection. Once the connection is made, LabVIEW waits to receive a command. LabVIEW executes the command and returns the results. The connection is then closed. LabVIEW repeats this entire process until it is shut down locally when a user presses a stop button on the front panel or remotely when it receives a command to shut the VI down.

This VI does not report errors. It might send back a response indicating that a command is invalid, but it does not display a dialog when an error occurs. Because a server might be unattended, consider carefully how the server should handle errors. You probably do not want a dialog box to be displayed because that requires user interaction at the server (someone would have to press the OK button). However, you might want LabVIEW to write a log of transactions and errors to a file or a string.

You can increase performance by allowing the connection to stay open so that you can receive multiple commands, but this action blocks other clients from connecting until the current client disconnects. If the protocol supports multiple simultaneous connections, you can restructure LabVIEW to handle multiple clients simultaneously, as shown in the following diagram.



The preceding diagram uses LabVIEW multitasking capabilities to run two loops simultaneously. One loop continuously waits for a connection. When a connection is received, it is added to a queue. The other loop checks each of the open connections and executes any commands that have been received. If an error occurs on one of the connections, the connection is disconnected. When the user aborts the server, all open connections are closed. This basic block diagram structure is a model used elsewhere in this manual to demonstrate how to implement a given protocol in LabVIEW.

21

TCP and UDP

This chapter discusses Internet Protocol (IP), User Datagram Protocol (UPD), Transmission Control Protocol (TCP), internet addresses, and examples of TCP client/server applications.

Overview

TCP/IP is a suite of communication protocols, originally developed for the Defense Advanced Research Projects Agency (DARPA). Since its development, it has become widely accepted and is available on a number of computer systems.

The name TCP/IP comes from two of the best known protocols of the suite, the Transmission Control Protocol (TCP) and the Internet Protocol (IP). TCP, IP, and the User Datagram Protocol (UDP) are the basic tools for network communication.

TCP/IP enables communication over single networks or multiple interconnected networks (internet). The individual networks can be separated by great geographical distances. TCP/IP routes data from one network or internet computer to another. Because TCP/IP is available on most computers, it can transfer information between diverse systems.

Internet Protocol (IP) transmits data across the network. This low-level protocol takes data of a limited size and sends it as a *datagram* across the network. Because it does not guarantee that the data will arrive at the other end, IP is rarely used directly by applications. Also, when you send several datagrams they sometimes arrive out of order or are delivered multiple times, depending on how the network transfer occurs. UDP, which is built on top of IP, has similar problems.

TCP is a higher-level protocol that uses IP to transfer data. TCP breaks data into components that IP can manage. It also provides error detection and ensures that data arrives in order without duplication. For these reasons, TCP is usually the best choice for network applications.

LabVIEW and TCP/IP

You can use the TCP/IP suite of protocols with LabVIEW on all platforms. LabVIEW has a set of TCP and UDP VIs that you can use to create client or server VIs.

Internet Addresses

Each host on an IP network has a unique 32-bit internet address. This address identifies the network on the internet to which the host is attached and the specific computer on that network. You use this address to identify the sender or receiver of data. IP places the address in the datagram headers so that each datagram is routed correctly.

One way of describing this 32-bit address is the IP dotted decimal notation, which divides the 32-bit address into four 8-bit numbers. The address is written as the four integers, separated by decimal points. For example, the following 32-bit address is written in dotted decimal notation as `132.13.2.30`.

```
10000100     00001101     00000010     00011110
```

Another way of using the 32-bit address is by names that are mapped to the IP address. Network drivers usually perform this mapping by consulting a local *hosts* file that contains name to address mappings or a larger database using the Domain Name System to query other computer systems for the address for a given name. Your network configuration dictates the exact mechanism for this process, which is known as *hostname resolution*.

Internet Protocol (IP)

Internet Protocol (IP) performs the low-level service of moving data between machines. IP packages data into components called *datagrams*. A datagram contains, among other things, the data and a header indicating the source and destination addresses. IP determines the correct path for the datagram to take across the network or internet and sends the data to the specified destination.

The original host may not know the complete path that the data will take. Using the header, any host on the network can route the data to the destination, either directly or by forwarding it to another host. Because some systems have different transfer capabilities, IP can fragment datagrams into smaller segments as necessary; when the data arrives at the destination, IP automatically reassembles the data into its original form.

IP makes a best-effort attempt to deliver data but cannot guarantee delivery. Also, because IP routes each separately, datagrams might arrive out of sequence. In fact, IP may deliver a single packet more than once if it is duplicated in transmission. IP does not determine the order of packets. Instead, higher-level protocols layered above IP order the packets and ensure reliable delivery. IP is rarely used directly because programs use TCP or UDP instead.

User Datagram Protocol (UDP)

UDP provides simple, low-level communication between processes on computers. Processes communicate by sending datagrams to a destination machine or port. IP handles the machine-to-machine delivery. Once on a machine, UDP moves the datagram to its destination port. If the destination port does not have a receiving process attached, the datagram is discarded. All of the delivery problems of IP are also present in UDP.

Typically, UDP is used in applications where reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic.

Using UDP

UDP is not a connection-based protocol like TCP. This means that a connection does not need to be established with a destination before sending or receiving data. Instead, the destination for the data is specified when each datagram is sent. The system does not report transmission errors.

You can use the UDP Open VI to open a port. A port is the location where data is sent. The number of simultaneously open UDP ports depends on the system. UDP Open returns a Network Connection refnum, an opaque token used in all subsequent operations pertaining to that port.

You can use the UDP Write VI to send data to a destination and the UDP Read VI to read it. Each write requires a destination address and port. Each read contains the source address and port. Packet boundaries are preserved. That is, a read never contains data sent in two separate write operations.

In theory, you should be able to send data packets of any size. If necessary, a packet is disassembled into smaller pieces and sent on its way. At their destination, the pieces are reassembled and the packet is presented to the requesting process. In practice, systems only allocate a certain amount of memory to reassemble packets. A packet that cannot be reassembled is

thrown away. The largest size packet that can be sent without disassembly depends on the network hardware.

When LabVIEW finishes all communications, calling the UDP Close VI frees system resources.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) ensures reliable transmission across networks, delivering data in sequence without errors, loss, or duplication. When you pass data to TCP, it attaches additional information and gives the data to IP, which puts the data into datagrams and transmits it. This process reverses at the receiving end, with TCP checking the data for errors, ordering the data correctly, and acknowledging successful transmissions. If the sending TCP does not receive an acknowledgment, it retransmits the data segment.

Using TCP

TCP is a connection-based protocol, which means that sites must establish a connection before transferring data. TCP permits multiple simultaneous connections.

You initiate a connection either by waiting for an incoming connection or by actively seeking a connection with a specified address. In establishing TCP connections, you have to specify both the address and a port at that address. A port is represented by a number between 0 and 65535. On UNIX systems, port numbers less than 1024 are reserved for privileged applications. Different ports at a given address identify different services at that address and make it easier to manage multiple simultaneous connections.

You can actively establish a connection with a specific address and port using the TCP Open Connection function. Using this function, you specify the address and port with which you want to communicate. If the connection is successful, the function returns a connection ID that uniquely identifies that connection. Use this connection ID to refer to the connection in subsequent function calls.

You can use two methods to wait for an incoming connection:

- With the first method, you use the TCP Listen VI to create a listener and wait for an accepted TCP connection at a specified port. If the connection is successful, the VI returns a connection ID and the address and port of the remote TCP.

- With the second method, you use the TCP Create Listener function to create a listener, and then use the Wait on Listener function to listen for and accept new connections. Wait on Listener returns the same listener ID that was passed to the function, as well as the connection ID for a connection. When you are finished waiting for new connections, you can use TCP Close to close a listener. You can not read from or write to a listener.

The advantage of using the second method is that you can cancel a listen operation by calling TCP Close. This is useful in the case where you want to listen for a connection without using a timeout, but you want to cancel the listen when some other condition becomes true (for example, when the user presses a button).

When a connection is established, you can read and write data to the remote application using the TCP Read and TCP Write functions.

Finally, use the TCP Close Connection function to close the connection to the remote application. If there is unread data and the connection closes, that data may be lost. Connected parties should use a higher-level protocol to determine when to close the connection. Once a connection is closed, you cannot read or write from it again.

TCP Versus UDP

If you are writing both the client and server and your system can use TCP/IP, then TCP is probably the best protocol to use because it is a reliable, connection-based protocol. UDP is a connectionless protocol with higher performance, but it does not ensure reliable transmission of data.

TCP Client Example

The following discussion is a generalized description of how to use the components of the Client block diagram model with the TCP protocol.

Use the TCP Open Connection function to open a connection to a server. You must specify the internet address of the server, as well as the *port* for the server. The address identifies a computer on the network. The port is an additional number that identifies a communication channel on the computer that the server uses to listen for communication requests. When you create a TCP server, you specify the port that you want the server to use for communication.

To execute a command on the server, use the TCP Write function to send the command to the server. You then use the TCP Read function to read back results from the server. With the TCP Read function, you must specify the number of characters you want to read. Because the length of the response might vary, this can be awkward. The server can have the same problem with the command because the length of a command can vary.

You can use the following methods to address differently sized commands:

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter and then read the number of characters specified by the size. This option is efficient and flexible.
- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. You then need to read data in small chunks until you get the terminating character.

Use the TCP Close Connection function to close the connection to the server.

Timeouts and Errors

The preceding section discussed communication protocol for the server. When you design a network application consider carefully what should happen if something fails. For example, if the server crashes, how would each of the client VIs handle it?

One solution is to make sure that each VI has a timeout. If something fails to produce results, after a certain amount of time the client will continue execution. In continuing, the client can try to reestablish execution or report the error. If necessary, it can gracefully shut down the client application.

TCP Server Example

The following discussion explains how you can use TCP to fulfill each component of the general server model.

No initialization is necessary with TCP, so you can leave out this step.

Use the TCP Listen VI to wait for a connection. You must specify the port that will be used for communication. This port must be the same port that the client will attempt to connect. For more information, see the *TCP Client Example* section in this chapter.

If a connection is established, read from that port to retrieve a command. As discussed in the TCP Client example, you must decide the format for commands. If commands are preceded by a length field, first read the length field, and then read the amount of data indicated by the length field.

Execution of a command should be protocol independent because it is done on the local computer. When finished, pass the results to the next stage, where they are transmitted to the client.

Use the TCP Write function to return results. As discussed in the *TCP Client Example* section, the data must be in a form that the client can accept.

Use the TCP Close Connection function to close the connection.

This step can be left out with TCP, because everything is finished after you close the connection.

TCP Server with Multiple Connections

TCP handles multiple connections easily. You can use the methods described in the preceding section to implement the components of a server with multiple connections.

Setup

Before you can use TCP/IP, you need to make sure that you have the right setup, which varies depending on the computer you use.

UNIX

TCP/IP support is built in. Assuming your network is configured properly, no additional setup for LabVIEW is necessary.

Macintosh

TCP/IP is built in to Macintosh operating system version 7.5 and later. To use TCP/IP with an earlier system, you need to install the MacTCP driver software, available from the Apple Programmer Developer Association (APDA). You can contact APDA at (800) 282-2732 for information on licensing the MacTCP driver. LabVIEW also works with Open Transport.

Windows 3.*x*

To use TCP/IP, you must install an ethernet board along with its low-level driver. In addition, you must purchase and install TCP/IP software that includes a Windows Sockets (WinSock) DLL conforming to standard 1.1. WinSock is a standard interface that enables application communication with a variety of network drivers. Several vendors provide network software that includes the WinSock DLL. Install the ethernet board, the board drivers, and the WinSock DLL according to the software vendor instructions.

Several vendors supply WinSock drivers that work with a number of boards. Contact the vendor of your board to find out if they offer a WinSock DLL you can use with the board. Install the WinSock DLL according to vendor instructions.

National Instruments recommends using the WinSock DLL provided by Microsoft for Windows for Workgroups. Prior to the release of this WinSock DLL, National Instruments tested a number of WinSock DLLs. These tests showed that many DLLs at that time did not fully comply with the standard, so consider trying a demo version of a DLL before you buy the real version. You can usually obtain a demo version from the manufacturer. Most demo versions are fully functional but expire after a certain amount of time.

Windows 95 and Windows NT

TCP support is built into Windows 95 and NT. You do not need to use a third-party DLL to communicate using TCP.

22

ActiveX Support

This chapter explains how to use LabVIEW as an ActiveX server and client.

With ActiveX automation, you can access properties and methods, which are usually grouped into objects, of one Windows application and use them in another Windows application. LabVIEW for Windows 3.*x* does not support ActiveX automation. ActiveX automation is supported only in Windows 95 and NT.

An application supports automation either as a server or a client. Applications that expose objects and provide methods for operating on those objects are ActiveX automation servers. Applications that use the methods exposed by another application are ActiveX automation clients.

LabVIEW can function both as an ActiveX server and an ActiveX client. LabVIEW can also display an ActiveX object on the front panel using the ActiveX container. For more information on ActiveX, refer to Chapter 16, *ActiveX Controls*, in the *G Programming Reference Manual*.

☞ **Note** ***For the purposes of this document, the term*** ActiveX ***refers to Microsoft Corporation's ActiveX technology, as well as to OLE technology.***

For general ActiveX information, refer to the Microsoft Developers Network documentation and *Inside OLE****, 2nd edition by Kraig Brockschmidt.***

ActiveX Automation Server Functionality

Because you can use LabVIEW as an ActiveX Automation server, other ActiveX-enabled applications (such as Microsoft Excel) can request properties and methods from LabVIEW and individual VIs.

To activate LabVIEW as an ActiveX server, select **Edit»Preferences» Server: Configuration**. The following dialog box appears:

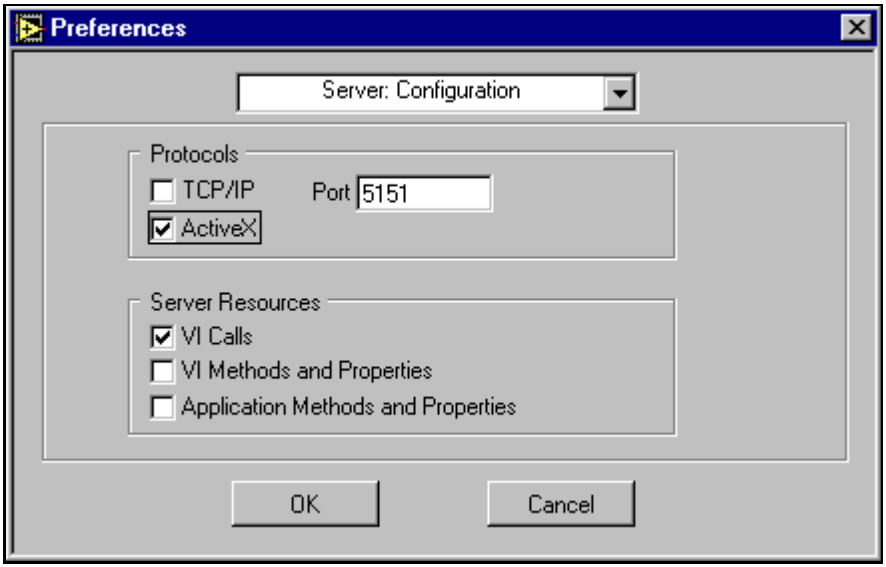


Figure 22-1. Preferences Dialog Box, Server Configuration

Select the **ActiveX** protocol. LabVIEW exports a createable class, Application, and a dispatch class, VirtualInstrument, to ActiveX. The progId `LabVIEW.Application` or `LabVIEW.Application.5` creates an application object. The Application method GetVIReference creates and returns a pointer to a Virtual Instrument object.

ActiveX Server Properties and Methods

See the LabVIEW *Online Reference* for detailed descriptions of the properties and methods of the exported classes.

You can find an example that illustrates how you can use properties and methods in `examples\comm` directory\`freqresp.xls`. This example uses a Visual Basic script macro to run a VI and tabulate the results.

ActiveX Automation Client Functionality

LabVIEW can act as an ActiveX Automation client, controlling other ActiveX Automation servers. LabVIEW can set and get properties and execute methods made available by ActiveX servers. A server exports information about its objects, methods, and properties through a Type Library file. A type library is normally created by the environment in which the servers were built. Refer to the documentation for each server application for more information.

Table 22-1 lists and describes the functions you can use for an ActiveX Automation client.

Table 22-1. Functions for ActiveX Automation Client Support

Function	Description
Automation Open	Selects an automation Class to be opened
Invoke Node	Executes functions of a class
Property Node	Sets or gets properties of a class
Automation Close	Closes an automation refnum

With the following steps, you can create a client application using C:

1. Get the IDispatch interface of the object whose methods you want to access.
2. Get the DispatchID of the method of that object.
3. Invoke the method using the Invoke functions of the IDispatch interface, packing all parameters into the parameter list.

To create a client application in LabVIEW, complete the following steps:

1. Use the Automation Open function to get an Automation refnum that uniquely defines the IDispatch interface.
2. Use the Invoke Node to execute a method belonging to that object. LabVIEW converts its data types to ActiveX variant if the server application requires data in that format.

The examples in the next section, *ActiveX Client Examples*, illustrates the use of these nodes.

ActiveX Client Examples

The following examples illustrate how to use the new ActiveX functions listed above.

Converting ActiveX Variant Data to G Data

The first example, shown in Figure 22-2, illustrates how to convert ActiveX Variant data to G data. In every ActiveX application, you must open an ActiveX automation refnum at the start and close the automation refnum at the end. In this example, you open the application object of Microsoft Excel and display the visible property using Property Node. The visible property is in ActiveX Variant format. The G Data function must convert the property information to a format supported by LabVIEW.

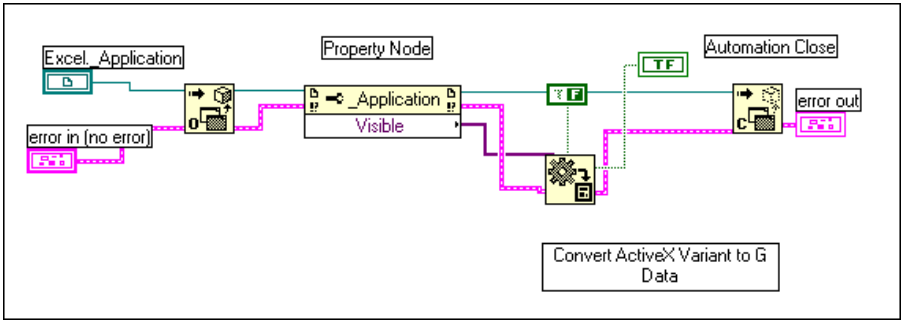


Figure 22-2. Block Diagram Displaying ActiveX Variant Data to G Data

To learn more about the property information of other applications, refer to the online help for that application. The LabVIEW *Online Reference* does not contain property information for other ActiveX-enabled applications.

Adding a Workbook to Microsoft Excel from LabVIEW

The second example, shown in Figure 22-3, adds a workbook to Microsoft Excel from LabVIEW. As stated earlier in this section, you must open every ActiveX application refnum with the Open Automation Refnum functions and close the ActiveX application with Close Automation Refnum. To add another workbook, you must have a refnum to a workbook. In this example, you open the Excel refnum with Open Automation Refnum and you access the workbook refnum with the Property Node. After you add a workbook to Excel, a refnum referring to that workbook is returned to LabVIEW. When you no longer need Excel open, close the Excel and workbook refnum.

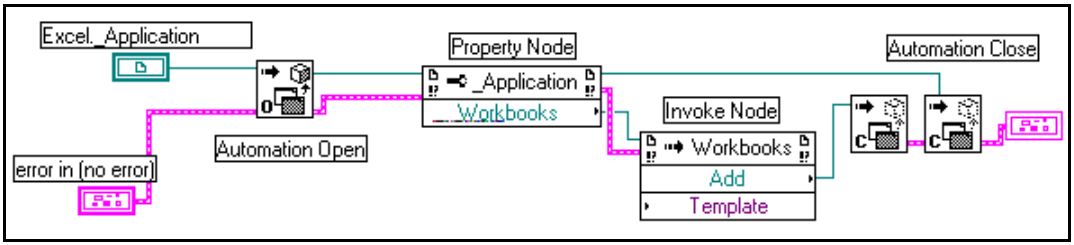


Figure 22-3. Adding a Workbook to Microsoft Excel

23

Using DDE

This chapter describes the LabVIEW VIs for Dynamic Data Exchange (DDE) for Windows 3.1, Windows 95, and Windows NT. These VIs execute DDE functions for sharing data with other applications that accept DDE connections.

DDE Overview

Dynamic Data Exchange (DDE) is a protocol for exchanging data between Windows applications.

In TCP/IP communications, applications open a line of communication and then transfer raw data. DDE works at a higher level, where applications send messages to each other to exchange information. One simple message is to send a command to another application. Most of the other messages deal with transferring data, where the data is referenced by name.

Both applications must be running, and both must give Windows their callback function address before DDE communication can begin. The callback function accepts any DDE messages that Windows sends to the application.

A DDE client initiates a conversation with another application (a DDE server) by sending a connect message. After establishing a connection, the client can send commands to the server and change or request the value of data that the server manages.

A client can request data from a server by a request or an advise. The client uses a request to ask for the current value of the data. If a client wants to monitor a value over a period of time, the client must request to be advised of changes. By asking to be advised of data value, the client establishes a link between the client and server through which the server notifies the client when the data changes. The client can stop monitoring the value of the data by telling the server to stop the advise link.

When the DDE communication for a conversation is complete, the client sends a close conversation message to the server.

DDE is most appropriate for communication with standard off-the-shelf applications such as Microsoft Excel.

With LabVIEW you can create VIs that act as clients to other applications (meaning they request or send data to other applications). You can also create VIs that act as servers that provide named information for access by other applications. As a server, LabVIEW does not use *connection*-based communication. Instead, you provide named information to other applications, which can then read or set the values of that information by name.

Services, Topics, and Data Items

With TCP/IP, you identify the process you want to talk to by its computer address and a port number. With DDE, you identify the application you want to talk to by referencing the name of a service and a topic. The server decides on arbitrary service and topic names. A given server generally uses its application name for the service, but not necessarily. That server can offer several topics that it is willing to communicate. With Excel, for example, the topic might be the name of a spreadsheet.

To communicate with a server, first find the names of the service and topic that you want to discuss. Then open a conversation using these two names to identify the server.

Unless you are going to send a command to the server, you usually work with data items that the server is willing to talk about. You can treat these as a list of *variables* that the server lets you manipulate. You can change variables by name, supplying a new value for the variable. Or, you can request the values of variables by name.

Examples of Client Communication with Excel

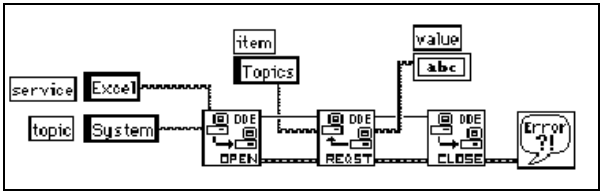
Each application that supports DDE has a different set of services, topics, and data items that it can talk about. For example, two different spreadsheet programs can take very different approaches to how they specify spreadsheet cells. To find out what a given application supports, consult the documentation that came with that application.

Microsoft Excel, a popular spreadsheet program for Windows, has DDE support. You can use DDE to send commands to Excel. You can also manipulate and read spreadsheet data by name. For more information on how to use DDE with Excel, refer to the *Microsoft Excel User's Guide 2*.

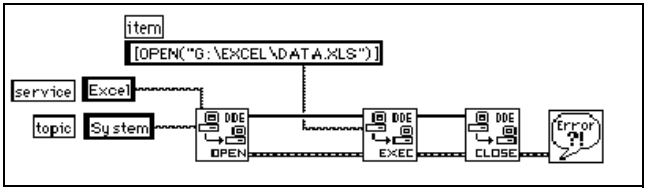
With Excel, the service name is `Excel`. For the topic, you use the name of an open document, such as spreadsheet document, or the word `System`.

If you use the name `System`, you can request information about the status of Excel or send general commands to Excel (commands that are not directed to a specific spreadsheet). For instance, for the topic `System`, Excel communicates about items such as `Status`, which has a value of `Busy` if Excel is busy, or `Ready` if Excel is ready to execute commands. `Topics` is another useful data item you can use when the topic is `System`. This item returns a list of topics about which Excel can communicate, including all open spreadsheet documents and the `System` topic.

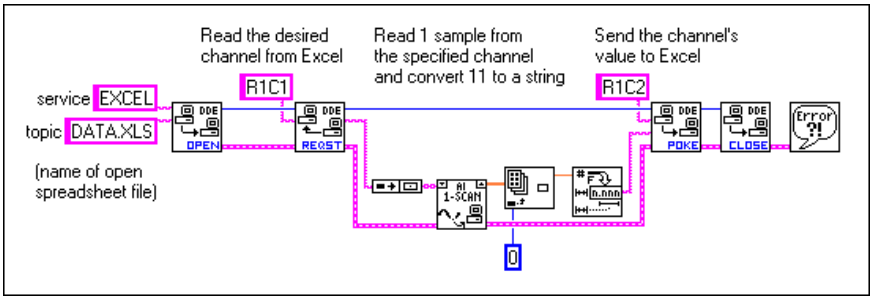
The following VI shows how you can use the Topics command in LabVIEW. The value returned is a string containing the names of the open spreadsheets and the word `System`.



Another way you can use the `System` topic with Excel is to instruct Excel to open a specific document. Use the `DDE Execute.vi` to send an Excel macro that instructs Excel to open the document, as shown in the following LabVIEW block diagram.



After you open a spreadsheet file, you can send commands to the spreadsheet to read cell values. In this case, your topic is the spreadsheet document name. The item is the name of a cell, a range of cells, or a named section of a spreadsheet. For example, in the following diagram LabVIEW can retrieve the value in the cell at row one column one. It then acquires a sample from the specified channel, and sends the resulting sample back to Excel.



LabVIEW VIs as DDE Servers

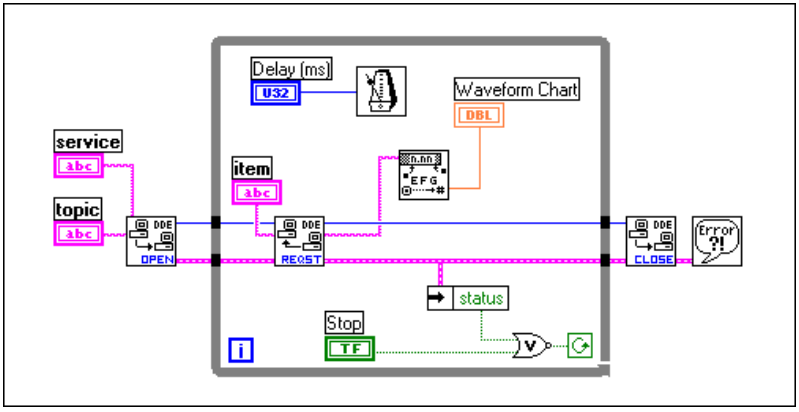
You can create LabVIEW VIs that act as servers for data items. The general concept is that a LabVIEW VI indicates that it is willing to provide information regarding a specific service and topic pair. LabVIEW can use any name for the service and topic name. It might specify the service name to be the name of the application (`LabVIEW`), and the topic name to be either the name of the Server VI, or a general classification for the data it provides, such as `Lab Data`.

The Server VI then registers data items for a given service that it will talk about. LabVIEW remembers the data names and their values, and handles communication with other applications regarding the data. When the server VI changes the value of data that is registered for DDE communication, LabVIEW notifies any client applications that have requested notification concerning that data. In the same way, if another application sends a `Poke` message to change the value of a data item, LabVIEW changes this value.

You cannot use the DDE Execute Command with a LabVIEW VI acting as a server. If you want to send a command to a VI, you must send the command using data items.

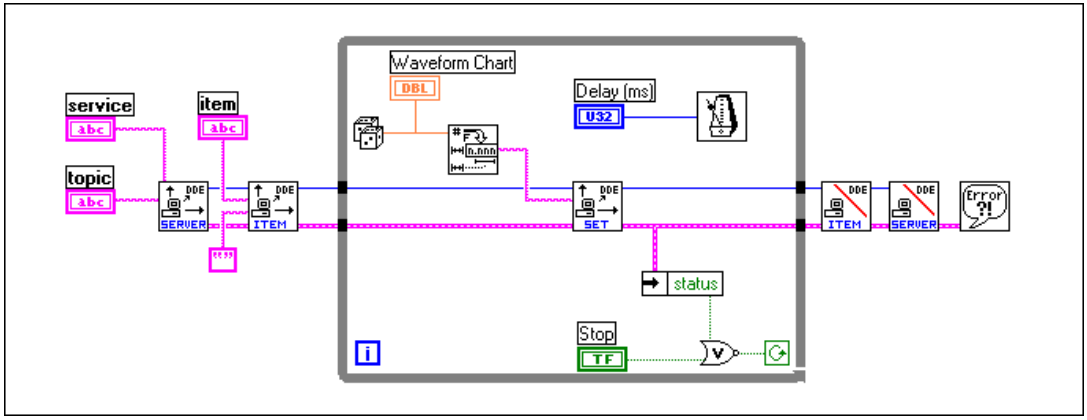
Also, notice that LabVIEW does not currently have anything like the System topic that Excel provides. The LabVIEW application is not inherently a DDE server to which you can send commands or request status information. However, you can use LabVIEW VIs to create a DDE server.

The following example shows how to create a DDE Server VI that provides data to other client applications. In this case, the data is a random number. You can easily replace the random number with real-world data from data acquisition boards or devices connected to the computer by GPIB, VXI, or serial connections.



The VI in the preceding diagram registers a server with LabVIEW. The VI registers an item that it is willing to provide to clients. In the loop, the VI periodically sets the value of the item. As mentioned earlier, LabVIEW notifies other applications that data is available. When the loop is complete, the VI finishes by unregistering the item and unregistering the server.

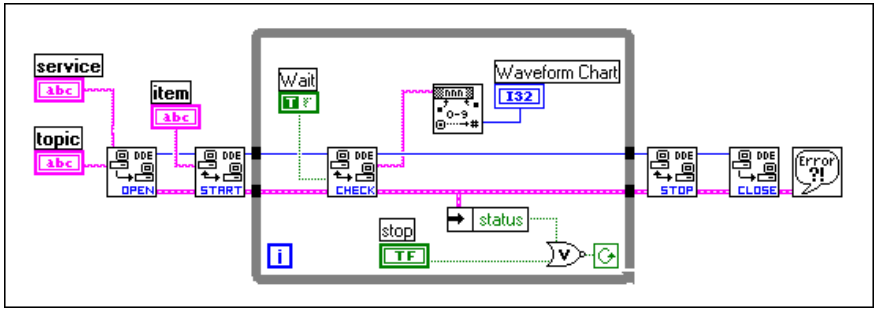
The clients for this VI can be any application that understands DDE, including other LabVIEW VIs. The following diagram illustrates a client to the VI shown in the previous diagram. It is important that the service, topic, and item names are the same as the ones used by the server.



Requesting Data Versus Advising Data

The previous client example used the DDE Request VI in a loop to retrieve data. With DDE Request, the data is retrieved immediately, regardless of whether you have seen the data before. If the server and the client do not loop at exactly the same rate, you can duplicate or miss data.

One way to avoid duplicating data is to use the DDE Advise VIs to request notification of changes in the value of a data item. The following diagram shows how you can implement this scheme.



In the preceding diagram, LabVIEW opens a conversation. It then uses the DDE Advise Start VI to request notification of changes in the value of a data item. Every time through the loop, LabVIEW calls the DDE Advise Check VI, which waits for a data item to change its value. When the loop is finished, LabVIEW ends the advise loop by calling the DDE Advise Stop VI, and closing the conversation.

Synchronization of Data

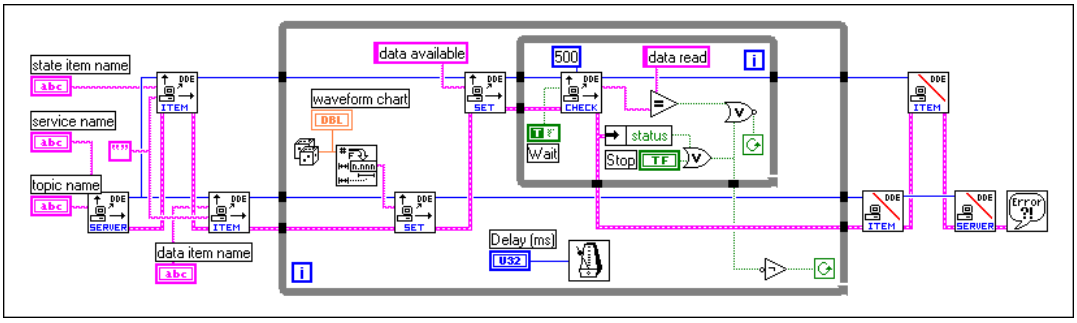
The client server examples in the preceding section work well for monitoring data. However, in these examples there is no assurance that the client receives all the data that the server sends. Even with the DDE Advise loop, if the client does not check for a data change frequently enough, the client can miss a data value that the server provided.

In some applications, missed data is not a problem. For example, if you are monitoring a data acquisition system, missed data may not cause problems when you are observing general trends. In other applications, you may want to ensure that no data is missed.

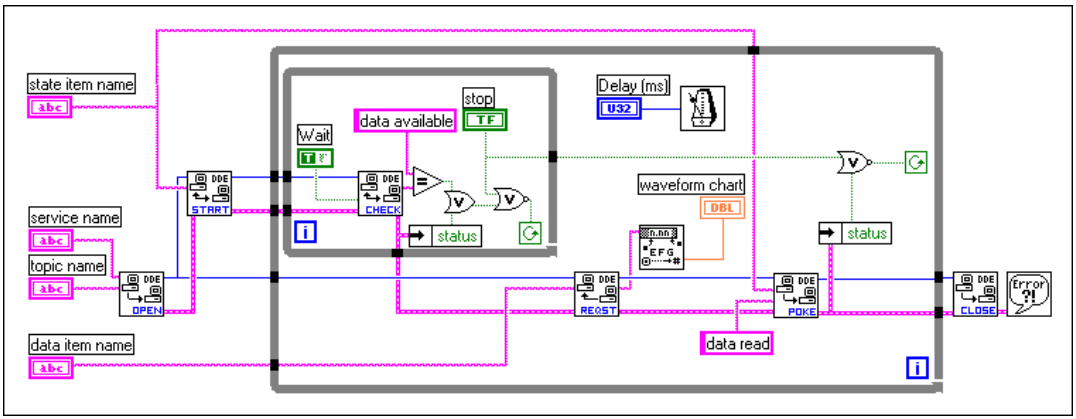
One major difference between TCP and DDE is that TCP queues data so that you do not miss it and you get it in the correct order. DDE does not provide this service.

In DDE, you can set up a separate item, which the client uses to acknowledge that it has received the latest data. You then update the acquired data item to contain a new point only when the client acknowledges receipt of the previous data.

For example, you can modify the server example shown in the *Requesting Data Versus Advising Data* section of this chapter to set a *state* item to a specific value after it has updated the acquired data item. The server then monitors the *state* item until the client acknowledges receipt of data. This modification is shown in the following block diagram.



A client for this server, as shown in the following diagram, monitors the state item until it changes to *data available*. At that point, the client reads the data from the acquired data item provided by the server, and then updates the state item to *data read* value.



This technique makes it possible to synchronize data transfer between a server and a single client. However, it has some shortcomings. First, you can have only one client. Multiple clients can conflict with one another. For example, one client might receive the data and acknowledge it before the other client notices that new data is available.You can build more complicated DDE diagrams to deal with this problem, but they quickly become awkward.

Another problem with this technique of synchronizing communication is that the speed of your acquisition becomes controlled by the rate at which you transfer data. You can address this issue by breaking the acquisition and the transmission into separate loops. The acquisition can queue data which the transmission loop would send. This is similar to the TCP Server example in which the server handles multiple connections.

If your application needs reliable synchronization of data transfer, you may want to use TCP/IP instead, because it provides queueing, acknowledgment of data transfer, and support for multiple connections at the driver level.

Networked DDE

You can use DDE to communicate with applications on the same computer or to communicate over the network with applications on different computers. To use networked DDE, you must be running Windows for Workgroups 3.1 or greater, Windows 95, or Windows NT. The standard version of Windows 3.1 does not support networked DDE.

Each computer under Windows for Workgroups has a network computer name. You configure this name using the Network control panel.

When you communicate over the network, the meaning of the service and topic strings change. The service name changes to indicate that you want to use networked DDE and includes the name of the computer you want to communicate with. The service name is of the following form:

`\\`*`computer-name`*`\ndde$`

You can supply any arbitrary name for the topic. You then edit the `SYSTEM.INI` file to associate this topic name with the actual service and topic that will be used on the remote computer. This configuration also includes parameters that configure the network connection. Following is an example of what this section would look like:

```
[DDE Shares]

topicname = appname, realtopic, ,31,,0,,0,0,0
```

The `topicname` is the name that your client VI uses for the topic. `Appname` is the name of the remote application. With networked DDE, this must be the same as the service name. `Realtopic` is the topic to use on the remote computer. The remaining parameters configure the way DDE works. Use the parameters as listed in the preceding example. The meaning of these parameters is not documented by Microsoft.

For example, if you want two computers running LabVIEW to communicate using networked DDE, the server needs to use `LabVIEW` for the service name, and a name, such as `labdata`, for the topic.

Assuming the server computer name is `Lab`, the client tries to open a conversation using the `\\Lab\ndde$` for the service. For the topic, the client can use a name of `remotelab`.

For this to work, you must edit the `SYSTEM.INI` file of the server computer to have the following line in the `[DDEShares]` section:

```
remotelab=LabVIEW,labdata,,31,,0,,0,0,0
```

For Windows NT, launch `DDEShare.exe`, which is located in the `winnt/system 32` directory. Choose **Shares»DDE Shares…** and then select **Add a Share…** to register the service name and topic name on the server.

Using NetDDE

NetDDE is built into Windows for WorkGroups 3.11, Windows 95 and Windows NT. It is also available for Windows 3.1 with an add-on package from WonderWare. If you are using Windows 3.1 with the WonderWare package, consult the WonderWare documentation on how to use NetDDE.

If you are using Windows for WorkGroups, Windows 95, or Windows NT, use the following instructions:

Server Machine

Windows for Workgroups

Add the following line to the `[DDE Shares]` section of the file `system.ini` on the server (application receiving DDE commands):

```
lvdemo = service_name,topic_name,,31,,0,,0,0,0
```

where

`lvdemo` can be any name.

`service_name` is typically the name of the application, such as `excel`.

`topic_name` is typically the specific file name, such as `sheet1`.

Enter other commas and numbers as shown.

Windows 95

☞ **Note** ***NetDDE is not automatically started by Windows 95. You need to run the program\WINDOWS\NETDDE.EXE. (This can be added to the startup folder so that it is always started.)***

To set up a NetDDE server on Windows 95:

- Run`\WINDOWS\REGEDIT.EXE`.
- In the tree display, open the folder `My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NetDDE\DDE Shares`.
- Create a new DDE Share by selecting **Edit»New»Key** and give it the name `lvdemo`.
- With the `lvdemo` key selected, add the required values to the share as follows. (For future reference, these keys are just being copied from the `CHAT$` share but `REDEGIT` does not allow you cut, copy, or paste

keys or values.) Use **Edit»New** to add new values. When you create the key, there will a default value named (`Default`) and a value of (`value not set`). Leave these values alone and add the following:

Table 23-1. Values to Add in Place of Default

Value Type	Name	Value
Binary	Additional item count	`00 00 00 00`
String	Application	`service_name`
String	Item	`service_name`
String	Password1	`service_name`
String	Password2	`service_name`
Binary	Permissions1	`1f 00 00 00`
Binary	Permissions2	`00 00 00 00`
String	Topic	`topic_name`

- Close `REGEDIT`.
- Restart the machine. (NetDDE must be restarted for changes to take effect.)

Windows NT

Launch `DDEShare.exe`, found in the `winnt\system32` directory. Select from the **Shares»DDE Shares»Add a Share...** to register the service name and topic name on the server.

Client Machine

On the client machine (application initiating DDE conversation) no configuration changes are necessary.

Use the following inputs to `DDE Open Conversation.vi`:

Service: `\\machine_name\ndde$`

Topic: `lvdemo`

where
`machine_name` specifies the name of the server machine

`lvdemo` matches the name specified in the `[DDE Shares]` section on the server.

Consider the examples `Chart Client.vi` and `Chart Server.vi` found in `examples\network\ddeexamp.llb`. To use those VIs to pass information between two computers using NetDDE, you should do the following:

Server Machine:

1. Do not modify any front panel values.
2. In the `system.ini` file of the Server machine, add the following line in the `[DDEShares]` section:
   `lvdemo = TestServer,Chart,,31,,0,,0,0,0`

Client Machine:

On the front panel, set the controls to the following:
Service = `\\machine_name\ndde$`
Topic = `lvdemo`
Item = `Random`

24

AppleEvents

This chapter describes AppleEvents, one form of Macintosh-only interapplication communication (IAC) through which Macintosh applications can communicate.

AppleEvents

AppleEvents is a Macintosh specific protocol that allows applications to communicate with each other. As with DDE, applications use a message to request actions or return information from other applications. An application can send a message to itself, an application on the same computer, or an application running on a computer elsewhere on the network. You can use AppleEvents to send commands to other applications, such as open or print, or to send data requests, such as spreadsheet information.

LabVIEW contains VIs for sending some of the commands common to most applications. The VIs are easy to use and do not require detailed knowledge of how AppleEvents work. The following list includes some of the ways you can use AppleEvents in LabVIEW applications:

- You can command LabVIEW to tell another application (even an application on another computer connected by a network) to perform an action. For example, LabVIEW can tell a spreadsheet program to create a graph. See the Sending AppleEvents section in this chapter for details.
- You can use an AppleScript program as a front end to instruct LabVIEW to run specific VIs.
- By sending instructions to perform specific operations, you can communicate with and control LabVIEW applications on other machines connected by a network. See the Sending AppleEvents section in this chapter for details.
- You can command LabVIEW to send messages to itself, instructing itself to load, run, and unload specific VIs. For example, in large applications where memory is tight, you can replace subVI calls with a utility VI (the AESend Open, Run, Close VI) and dynamically load, run, and unload the VIs.

These VIs use the low-level AESend VI to send AppleEvents. Apple has defined a large *vocabulary* for messages to help standardize AppleEvent communication. You can combine *words* in this vocabulary to build complex messages. You can use this VI to send arbitrary AppleEvents to other applications. However, creating and sending AppleEvents at this level is complicated and requires detailed understanding of AppleEvents. See *Inside Macintosh* and the *AppleEvent Registry.*

Sending AppleEvents

The **Communication** subpalette of the **Functions** palette contains VIs for sending AppleEvents. With these VIs, you can select a target application for an AppleEvent, create AppleEvents, and send the AppleEvents to the target application.

The **AppleEvent VIs** palette of the **Communication** subpalette contains VIs that send specific AppleEvent messages. These VIs let you send several standard AppleEvents (Open Document, Print Document, and Close Application) and all the LabVIEW custom AppleEvents. These high-level VIs require little understanding of AppleEvent programming details. Their diagrams are good examples of creating and sending AppleEvents.

You can use the low-level AESend VI if you want to send an AppleEvent for which LabVIEW provides no VI. The **AppleEvent VIs** palette of the **Communication** subpalette also contains VIs that can help you create an AppleEvent. However, creating and sending an AppleEvent at this level requires detailed understanding of AppleEvents as described in *Inside Macintosh, Volume VI*, and the *AppleEvent Registry.*

Client Server Model

You cannot use the AppleEvent VIs to create LabVIEW diagrams that behave as servers. The VIs are used to send messages to other applications. If you need diagram-based server capabilities, you must use TCP or PPC.

LabVIEW itself acts as an AppleEvent server, in that it understands and responds to a set of AppleEvents. Specifically, using AppleEvents you can instruct LabVIEW to open VIs, print them, run them, and close them. You can ask LabVIEW whether a given VI is running. You can also tell LabVIEW to quit.

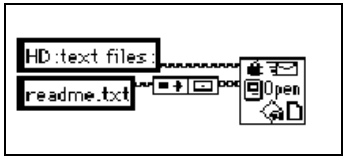
Using these server capabilities, you can instruct other LabVIEW applications to run VIs and control LabVIEW remotely. You can also command LabVIEW to send messages to itself, instructing the loading of specific VIs. For example, in large applications where memory is limited,

you can replace subVI calls with calls to the AESend Open, Run, Close VI to load and run VIs as necessary. Notice that when you run a VI this way its front panel opens, just as if you had selected **File»Open...**.

AppleEvent Client Examples

Launching Other Applications

To send a message to an application, that application must be running. You can use the AESend Finder Open VI to launch another application. This VI sends a message to the Finder. The Finder is, in itself, an application that understands a limited number of AppleEvents. The following simple example shows how you can use AppleEvents to launch Teach Text with a specific text file.



If the application is on a remote computer, then you must specify the location of that computer. You can use inputs to the AESend Finder Open VI to specify the network zone and the server name of the computer with which you want to communicate. If the network zone and server name are not specified, as in the preceding application, they default to those of the current computer.

Notice that if you try to send messages to another computer, you are automatically prompted to log onto that computer. There is no method for avoiding this prompt, because it is built into the operating system. This can cause problems when you want your application to run on an unattended computer system.

Sending Events to Other Applications

Once an application is running, you can send messages to that application using other AppleEvents. Not all applications support AppleEvents, and those that do may not support every published AppleEvent. To find out which AppleEvents an application supports, consult the documentation that comes with that application.

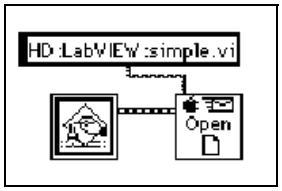
If the application understands AppleEvents, you call an AppleEvent VI with the Target ID for the application. A Target ID is a cluster that describes a target location on the network (zone, server, and supporting application).

You do not need to worry about the exact structure of this cluster because LabVIEW provides VIs that you can use to generate a Target ID.

There are two ways to create a Target ID. You can use the Get Target ID VI to programmatically create a Target ID based on the application name and network location. Or, you can use the PPC Browser VI, which displays a dialog box listing network applications that are aware of AppleEvents. You interactively select from this list to create a Target ID.

You can also use the PPC Browser VI to find out if another application uses AppleEvents. If you run the VI and select the computer that is running the application, the dialog box lists the application if it is AppleEvent aware.

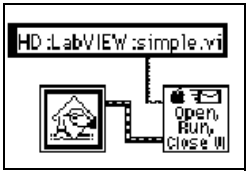
In the following diagram, LabVIEW interactively selects an AppleEvent-aware application on the network and tells it to open a document. In this case, LabVIEW is telling the application to open a VI.



Dynamically Loading and Running a VI

The AESend Open, Run, Close VI sends messages asking LabVIEW to run a VI. First, it sends the Open Document Message and LabVIEW opens a VI. Then, the Open Run Close VI sends the LabVIEW Run VI message and LabVIEW runs the specified VI. Next, Open Run Close sends the VI Active? message, and LabVIEW returns the status of a specified VI until the VI is no longer running. Finally, the VI sends the Close VI message.

Assuming the target LabVIEW is on another computer, you could use the following diagram to load and run the VI. If you are sending it to the current LabVIEW, you do not need the PPC Browser VI.



25

Program-to-Program Communication

This chapter describes program-to-program communication (PPC), a low-level form of Apple interapplication communication (IAC) by which Macintosh applications send and receive blocks of data.

Introduction to PPC

Program-to-Program Communication (PPC) is a Macintosh protocol for transferring blocks of data between applications. You can use it to create VIs that act as clients or servers. Although supported by all Macintoshes running System 7.*x*, it is not commonly used by most Macintosh applications. Instead, most Macintosh applications use AppleEvents, a high-level protocol for sending commands between applications, to communicate.

Although PPC is not as commonly supported as AppleEvents, it does provide some advantages. Because it is at a lower level, it provides better performance than AppleEvents. Also, in LabVIEW you can create VIs that use PPC to act as clients or servers. You cannot create diagrams that act as AppleEvent servers.

LabVIEW VIs can use PPC to send and receive large amounts of information between applications on the same computer or different computers on a network. For two applications to communicate with PPC, they must both be running and prepared to send or receive information.

PPC is similar in structure to TCP, in terms of both server and client applications. The PPC method for specifying a remote application is different from the TCP method. Other than that, the two protocols provide similar performance and features. Both protocols handle queueing and reliable transmission of data. You can use both protocols with multiple open connections.

When deciding between TCP and PPC, consider the platforms you plan to run your VIs on and the platforms with which you will communicate. If your application is Macintosh only, PPC is a good choice because it is built into the operating system. TCP is built into Macintosh operating system version 7.5 and later. To use TCP with an earlier system, you must buy a separate TCP/IP driver from Apple. If buying the separate driver is not an issue, then you may want to use TCP because the TCP interface is simpler than PPC. PPC uses some fairly complicated data structures to describe addresses.

If your application must communicate with other platforms or run on other platforms, then you should use TCP/IP.

Ports, Target IDs, and Sessions

To communicate using PPC, both clients and servers must open ports to use for subsequent communication. The Open Port VI opens the port using a cluster that contains, among other things, the name that you want to use for the port. Ports are used to distinguish between different services that an application provides. Each application can have multiple ports open simultaneously.

Each port can support several simultaneous sessions or conversations. To open a session, a client uses a Target ID indicating the location of the server. PPC uses the same type of Target ID that the AppleEvent VIs use. You can use the PPC Browser or the Get Target ID VIs to generate the Target ID for the remote application.

A server waits for clients to attempt to open a session by using the PPC Inform Session VI. The server can accept or reject the session by using the PPC Accept Session VI. A client can attempt to open a session with a server by using the PPC Start Session VI.

After the session is started, you can use the PPC Read and PPC Write VIs to transfer data. You can close a session using PPC End Session, and you can close a port using the PPC Close Port VI.

PPC Client Example

The following discussion explains how you can use PPC to fulfill each component of the general Client model.

Use the PPC Open Connection and PPC Open Session VIs to open a connection to a server. This requires that you specify the Target ID of the server, which you can get by using either the PPC Browser VI or the Get Target ID VI. The end result is a port refnum and a session refnum, which are used to communicate with the server.

To execute a command on the server, use the PPC Write VI to send the command to the server. Next, use the PPC Read VI to read the results from the server. With the PPC Read VI, you must specify the number of characters you want to read. As with TCP, this can be awkward because the length of the response can vary. The server can have a similar problem because the length of a command may vary.

Following are several methods that address the problem of differently sized commands. These methods can also be used with TCP.

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter, and then read the number of characters specified by the size. This option is efficient and flexible.
- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. You then need to read data in small chunks until you get the terminating character.

Use the PPC Close Session and PPC Close Connection VIs to close the connection to the server.

PPC Server Example

The following discussion explains how you can use PPC to fulfill each component of the general Server model.

Use PPC Open Port in the initialization phase to open a communication port.

Use the PPC Inform Session VI to wait for a connection. With PPC, you can either automatically accept incoming connections or choose to accept or reject the session by using the PPC Accept Session VI. This process of waiting for a session and then approving the session allows you to screen connections.

When a connection is established, you can read from that session to retrieve a command. As was discussed in the *PPC Client Example* section, you must decide the format for commands. If commands are preceded by a length field, then you need to first read the length field and then read that amount of data.

Because it is something done on the local computer, execution of a command should be protocol independent. When finished, you pass the results to the next stage, where they are transmitted to the client.

Use the PPC Write VI to return the result. As discussed in the *PPC Client Example* section, the data must be formatted in a form that the client can accept.

Use the PPC Close Session VI to close the connection.

When the server is finished, use the PPC Close Port VI to close the port that you opened in the initialization phase.

PPC Server with Multiple Connections

PPC handles multiple sessions and multiple ports easily. The methods for implementing each component of a server, as described in the preceding section, also work for a server with multiple connections. Figure 25-1 illustrates the order in which you use the PPC VIs.

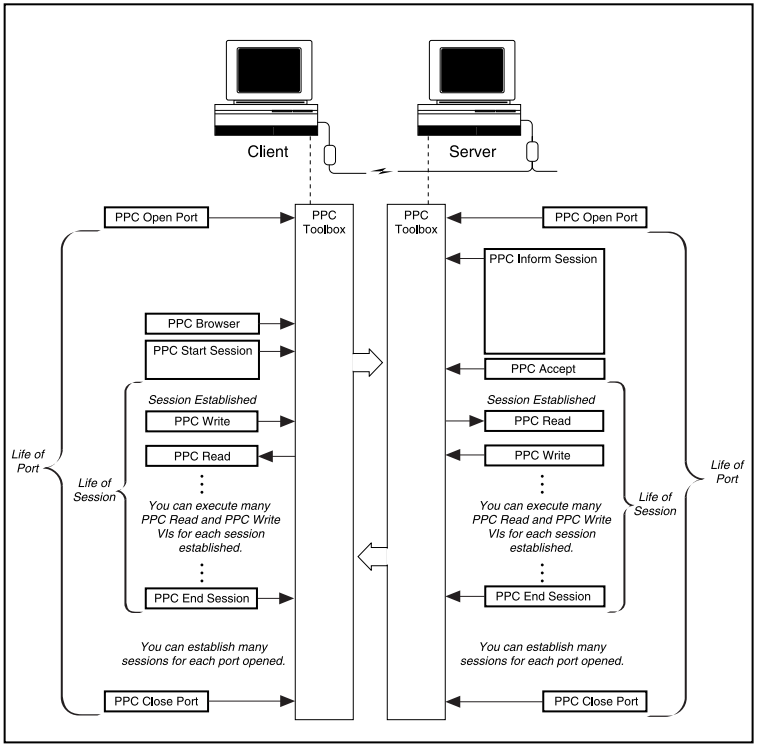


Figure 25-1. PPC VI Execution Order (Used by Permission of Apple Computer, Inc.)

Part V

Advanced G Programming

This section contains information about customizing VIs, configuring front panel objects programmatically, and designing complex applications.

Part V, *Advanced G Programming*, contains the following chapters.

- Chapter 26, *Customizing VIs*, explains how you can customize your VIs. It also contains activities that illustrate how to use the **VI Setup…** and the **SubVI Node Setup…** options to customize the appearance and execution behavior of a VI when it is running.
- Chapter 27, *Front Panel Object Attributes*, describes objects called attribute nodes, which are special block diagram nodes that control the appearance and functional characteristics of controls and indicators.
- Chapter 28, *Program Design*, explains techniques to use when creating programs and offers programming-style guidelines.
- Chapter 29, *Where to Go from Here*, provides information about resources you can use to create your applications successfully.

Note: **(Windows 3.1)** ***You must save the VIs you create in Part V in VI libraries. VI libraries enable you to use file names that are longer than 8 characters. Also, the VIs needed for the activities in Part V are located in the VI library*** `LabVIEW\Activity\Activity.llb`***. Refer to the*** *Saving VIs* ***section in Chapter 2,*** *Editing VIs****, of the*** *G Programming Reference Manual* ***for more information on VI libraries.***

In addition to programmatically configuring front panel objects, you also can configure and control VIs and LabVIEW itself programmatically. For example, you can change the appearance of a VI, change the execution behavior of a VI, change the print settings for LabVIEW, and determine all the VIs loaded into memory. Some of the options you can set interactively in **Preferences** and **VI Setup…** you also can set programmatically. The options available in **Preferences** are considered application settings because they affect all VIs in LabVIEW. The **VI Setup…** options are

VI settings because they only affect one VI and all instances in which you use it as a subVI. Besides changing the configuration of a VI and LabVIEW, you also can perform actions, or methods. For example, you can reinitialize all front panel objects to their default value.

Another example of programmatic control is dynamically loading and calling a VI. If you have a large application that has many subVIs, all the subVIs are loaded into memory when you load the top-level VI. Loading all subVIs into memory might not be practical for large applications. Instead, you can load and call subVIs dynamically using the Call By Reference node.

Refer to Chapter 21, *VI Server*, of the *G Programming Reference Manual* and the LabVIEW *Online Reference* for more information about programmatic configuration and control of VIs and LabVIEW.

In addition to configuring and controlling VIs and LabVIEW on a local machine, you can also configure and control VIs and LabVIEW on a remote machine through a TCP/IP network. Refer to Chapter 7, *Customizing Your Environment*, in the *G Programming Reference Manual* for more information. You also can configure and control VIs and LabVIEW through another ActiveX application. Refer to Chapter 22, *ActiveX Support*, in this manual for information about ActiveX support in LabVIEW.

26

Customizing VIs

This chapter explains how you can customize your VIs. It also contains activities that illustrate how to accomplish the following tasks:

- Use the **VI Setup…** option
- Use the **SubVI Node Setup…** option

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

How Do You Customize a VI?

You can configure VI execution in several ways. You access these options by popping up on the icon pane in the upper-right corner of the front panel and choosing **VI Setup…**.

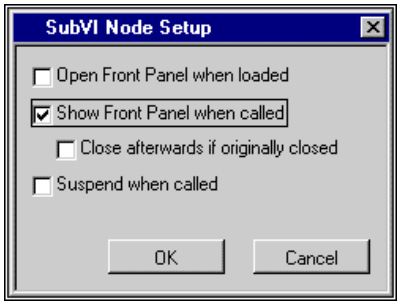
A VI Setup dialog box appears showing setup options for execution of the VI, appearance of the panel, documentation, and menu bar appearance at run time. You can learn how to use these options in Activity 26-1. For more detailed information, see Chapter 6, *Setting Up VIs and SubVIs*, in the *G Programming Reference Manual*.

Set Window Options

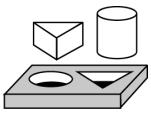
Use **Window Options** to control the appearance of the VI when running. To switch from **Execution Options** to **Window Options**, click the downward pointing arrow in the menu bar.

SubVI Node Setup

You also can configure how a subVI executes. The configuration options are available by popping up on the subVI icon in the block diagram of the calling VI, and choosing **SubVI Node Setup…**. The following illustration shows the SubVI Node Setup dialog box.



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



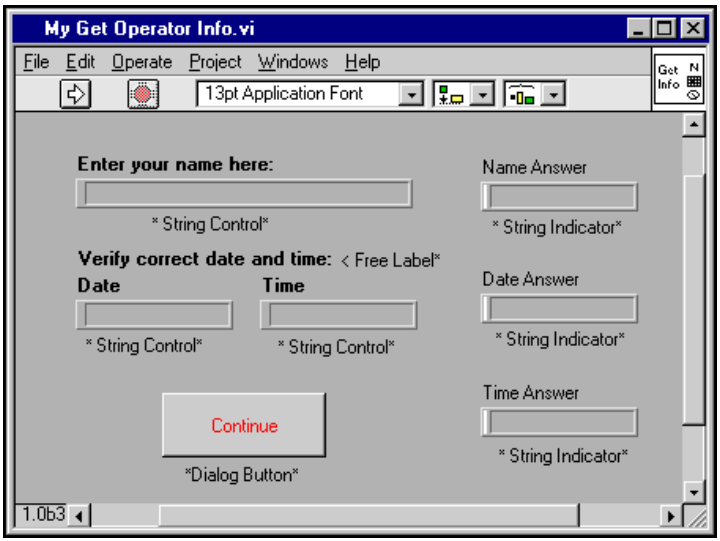
Activity 26-1. Use Setup Options for a SubVI

Your objective is to build a VI that prompts the operator to enter information.

You will create a VI that launches a dialog box to obtain information from the user upon execution. Once the user enters the information and clicks a button, the dialog box disappears.

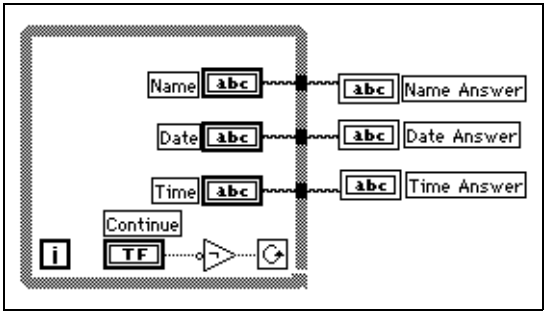
Front Panel

1. Open a new front panel and add the string controls and the button shown in the following illustration.



Block Diagram

2. Build the block diagram shown in the following illustration.

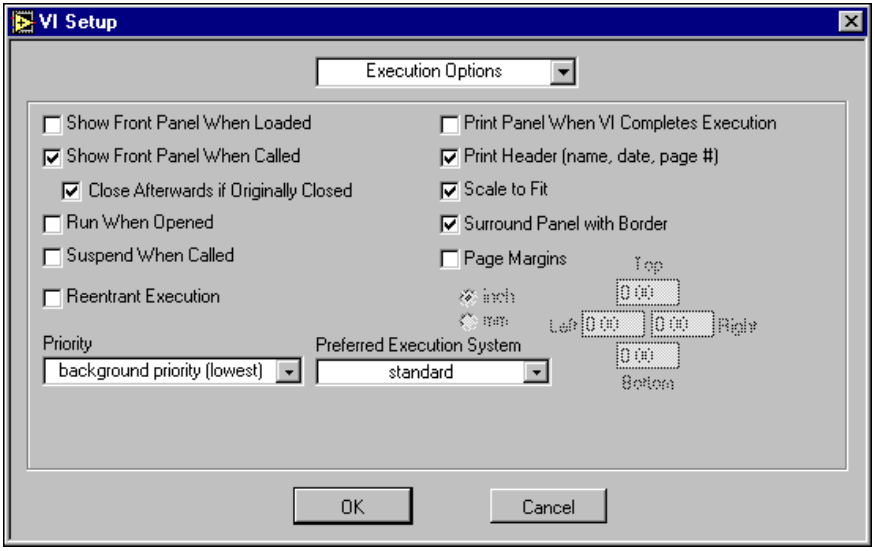


3. Create the icon for the VI as shown at left. To access the Icon Editor, pop up on the icon pane of the front panel and select **Edit Icon**.
4. Switch to the connector pane by popping up on the icon pane and selecting **Show Connector**.

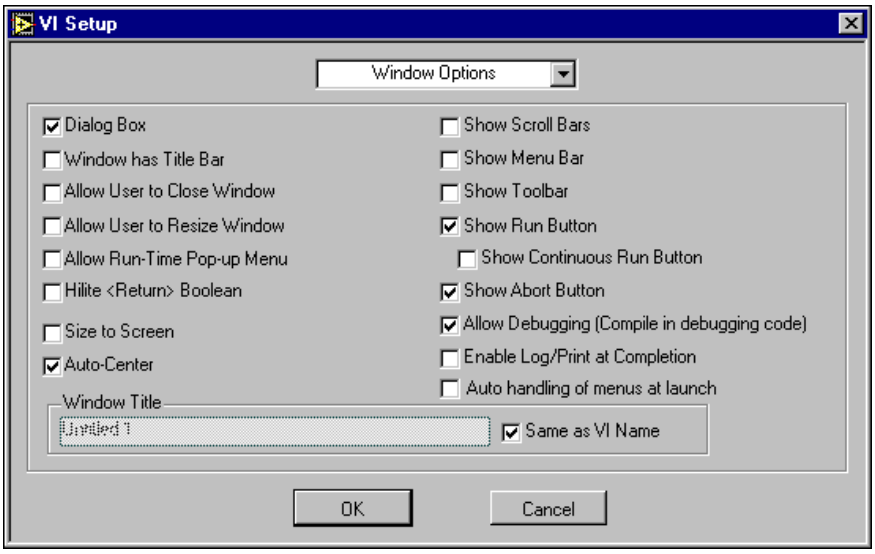
5. Build the connector. Notice that the default connector pane is not the same as the connector pane illustrated to the left. To create the correct connector pane, choose **Patterns** from the pop-up menu on the connector. Choose the pattern with three inputs and two outputs. Then choose **Flip Horizontal**. Now you can connect the **Date** and **Time** controls to the two connectors on the left side of the icon, and the

Name Answer, **Date Answer**, and **Time Answer** indicators to the three connectors on the right side of the icon, as shown in the following illustration. After creating the connector, return to the icon display.

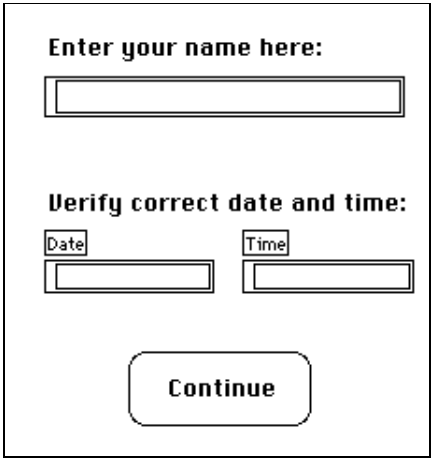
6. Save the VI as `Get Operator Info.vi` in the `LabVIEW\Activity` directory.
7. Now you can customize the VI with the **VI setup** options to make it look like a dialog box.
   a. Pop up on the icon and select **VI Setup**. Configure the **Execution Options** as shown in the following illustration.



   b. Select **Window Options** and make the selections shown in the following illustration.



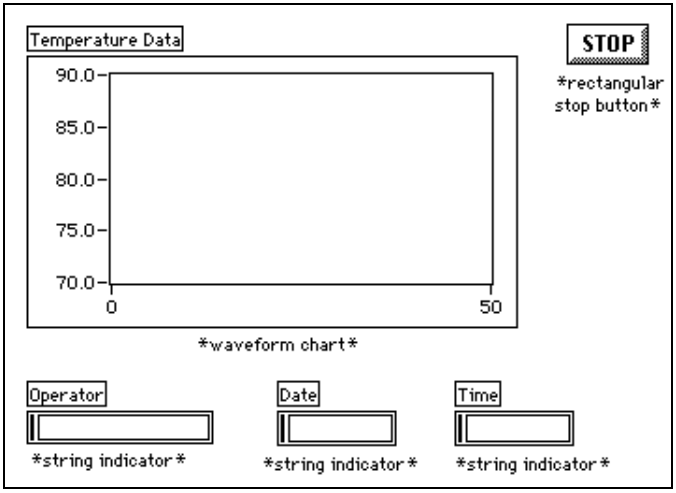
8. After you select the **VI Setup** options, resize the front panel as shown in the following illustration so you do not see the three string indicators.



9. Save and close the VI. Now you can use this VI as a subVI.

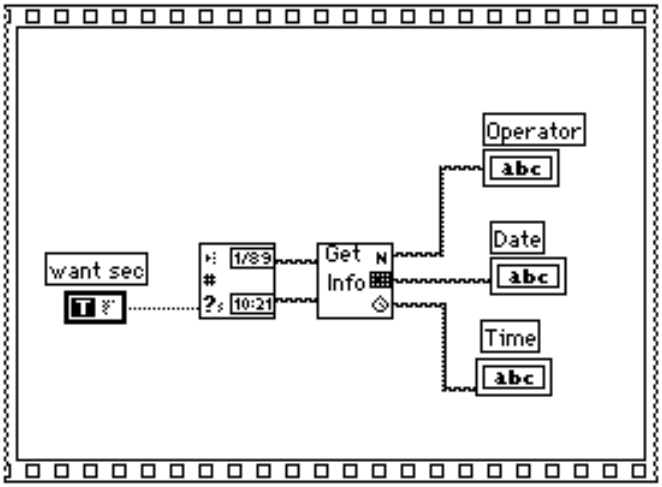
Front Panel

10. Open a new front panel.
11. Place a Waveform Chart (**Controls»Graph**) on the front panel and label it `Temperature Data`.
12. Modify the scale of the chart so that the upper limit is set to `90.0` and the lower limit is set to `70.0`. Pop up on the chart and choose **Show»Legend** to hide the legend. Pop up on the chart again and choose **Show»Palette** to hide the palette.
13. Build the rest of the front panel as shown in the following illustration.



Block Diagram

14. Create a Sequence structure and add the following objects to frame 0, as shown in the following illustration.



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

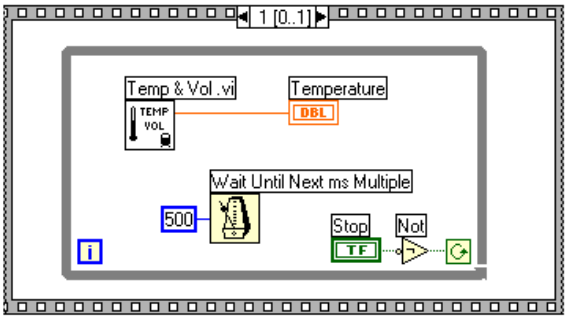
Get Operator Info VI (**Functions»Select a VI…** from the `LabVIEW\Activity` directory)—Opens its front panel and prompts the user to enter a name, the date, and the time.

Boolean constant (**Functions»Boolean**)—Controls whether the input date and time string are TRUE. To set this option to TRUE, click the constant with the Operating tool.

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

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

17. Add the objects shown in the following illustration.



Temp & Vol VI (**Functions»Select a VI…** from the `LabVIEW\Activity` directory)—Returns one temperature measurement from a simulated temperature sensor.

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

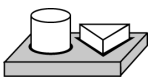
Numeric constant (**Functions»Numeric**)—You also can pop up on the Wait Until Next Tick Multiple function and select **Create Constant** to create and wire the numeric constant automatically. The numeric constant delays execution of the loop for 500 ms (0.5 seconds).

Not function (**Functions»Boolean**)—Inverts the value of the **STOP** button so that the While Loop executes repeatedly until you click **STOP**.

18. Save the VI as `Pop-up Panel Demo.vi` in the `LabVIEW\Activity` directory.
19. Run the VI. The front panel of the Get Operator Info VI opens and prompts you to enter your name, the date, and the time. Click the **Continue** button to return to the calling VI. Then temperature data is acquired until you click the **STOP** button.

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

20. Close all windows.



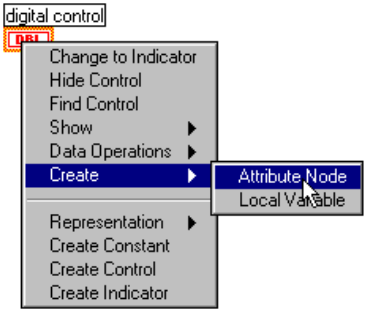
End of Activity 26-1.

27

Front Panel Object Attributes

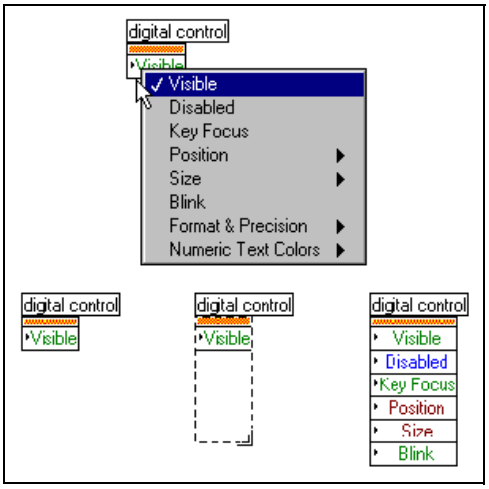
This chapter describes objects called attribute nodes, which are special block diagram nodes that control the appearance and functional characteristics of controls and indicators.

With attribute nodes, you can set attributes such as display colors, visibility, position, blinking, and many more. To create an attribute node, select **Create»Attribute Node** from the pop-up menu of the front panel object or from the terminal in the block diagram, as shown in the following illustration.



Initially, the attribute node displays a single characteristic. You can expand the node to display multiple characteristics. To expand the node, select the attribute node with the Positioning tool. Place your cursor over the node near the bottom-right corner, and when your cursor changes to a frame drag it to create the desired number of characteristics. Then you can change

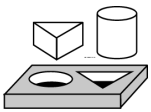
attributes by clicking the node with the Operating tool and choosing the new attribute from the pop-up menu, as shown in the following illustration.



Because there are many different attributes for front panel objects, you can use the Help window to display the descriptions, data types, and acceptable values of attributes. Access the Help window by selecting **Help» Show Help**.

For more information about accessing help in LabVIEW, see *Getting Help* in Chapter 1, *Introduction to G Programming*, of the *G Programming Reference Manual*.

With attribute nodes, you can assign characteristics or read the current state of an attribute by popping up on the attribute and selecting **Change to Read**.

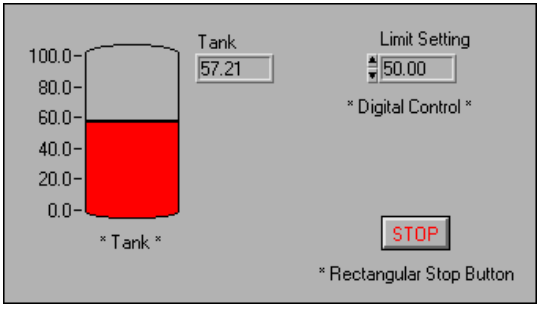


Activity 27-1. Use an Attribute Node

Your objective is to create a VI that indicates a high limit condition using attribute nodes. You will use the* Fill Color *attribute of a* `Tank indicator` *to indicate whether a randomly generated tank level has gone above the user-defined limit.

Front Panel

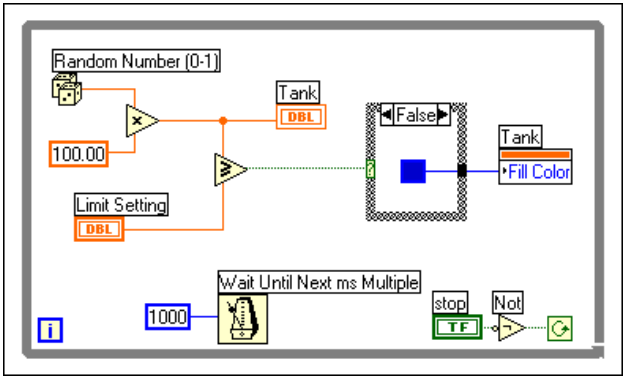
1. Open a new front panel and create it as shown in the following illustration.



2. Rescale the tank from `0.0` to `100.0`.
3. Set the default **Limit Setting** to `50.00`.

Block Diagram

4. Create the block diagram as shown below.



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

Random Number Generator (**Functions»Numeric**)—Generates raw data between 0 and 1 to fill the tank on your front panel. You multiply this value by 100 to create a value between 0 and 100.

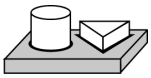
Greater or Equal? (**Functions»Comparison**)—Compares the raw data to the **Limit Setting** input. If the value is greater than or equal to the limit input, a TRUE value is passed to the Case Structure.

Attribute Node (Pop up on the Tank terminal)—Select **Create» Attribute Node** from the Tank terminal. Pop up on the attribute and choose **Select»Fill Color**.

Color Box Constant (**Functions»Numeric»Additional Numeric Constants**)—Wire this constant to define a red color to **Fill Color** in the TRUE case and a blue color in the FALSE Case. Click on the constant with the Operating tool to select the color.

Wait Until Next ms Multiple (**Functions»Time & Dialog**)—Wire a numeric constant of 1000 to execute the loop every second.

5. Run the VI. The level of the tank is compared to the **Limit Setting** control. If the tank value is greater than or equal to the **Limit Setting** value, the tank turns red. If the data falls below the limit, the tank turns blue.
6. Save the VI as `Tank Limit.vi` in the `LabVIEW\Activity` directory.



End of Activity 27-1.

28

Program Design

Now that you are familiar with many aspects of G programming, you need to apply that knowledge to develop your own applications. This chapter suggests some techniques to use when creating programs and offers programming-style recommendations.

Use Top-Down Design

When you have a large project to manage, incorporate *top-down design*. G has an advantage over other programming languages with respect to top-down design because you can start with the final user interface then animate it.

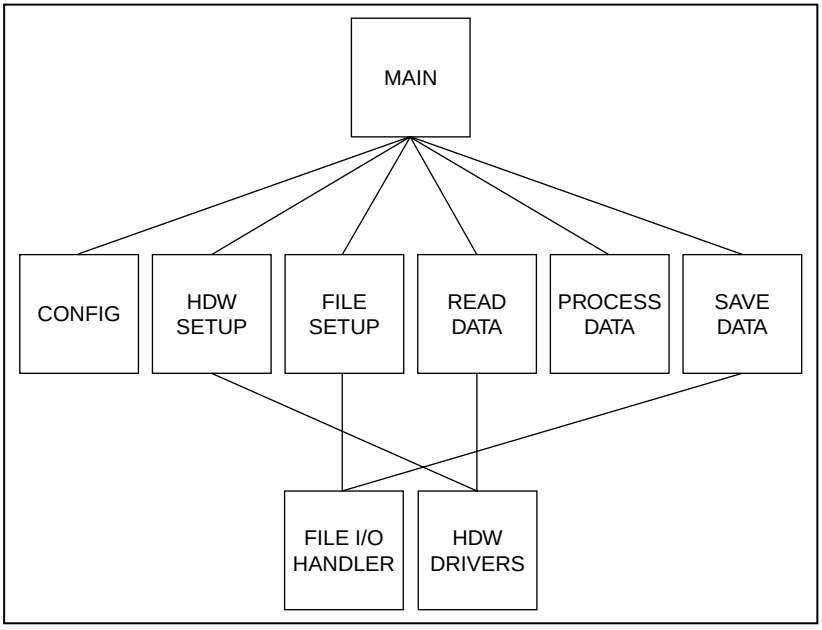
Make a List of User Requirements

Create a list of the panels with which the user can interact, the number and type of controls and indicators for these panels, the need for real-time analysis, data presentation, and so on. Next, create mock-up front panels you can show to the prospective users (or manipulate yourself, if you are the user). Think about and discuss functions and features. Use this interactive process to redesign the user interface as necessary. You might need to do some low-level research at this early stage to be certain you can meet specifications.

Design the VI Hierarchy

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

Divide the task to be accomplished into manageable, logical pieces. As the following flowchart illustrates, you can expect several major blocks in one form or another for every data acquisition system.



In some cases you might not need all these blocks or you might need different blocks. For example, some applications do not include any file I/O operations. Alternatively, you might need additional blocks, such as blocks representing user prompts. Your main objective is to divide your programming task into high-level blocks that you can manage easily.

After you determine the high-level blocks you need, try to create a block diagram that uses those high-level blocks. For each block, create a new *stub VI* (a nonfunctional prototype representing a future subVI). For this stub VI, create an icon as well as a front panel that contains the necessary inputs and outputs. You do not have to create a block diagram for this VI yet. Instead, see if this stub VI is a necessary part of your top-level block diagram.

After you assemble a group of stub VIs, try to understand, in general terms, the function of each block and how each block provides the desired results. Ask yourself whether any given block generates information that a subsequent VI needs. If so, make certain that the sketch for your top-level block diagram contains wires to pass the data between VIs.

Try to avoid using unnecessary global variables because they hide the data dependency between VIs. As your system gets larger, it becomes difficult to debug if you depend on global variables as your method for transferring information between VIs.

Create the Program

Now you are ready to create the program in G:

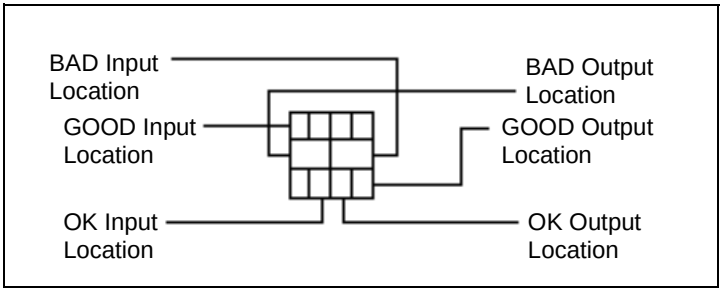
- Use a modular approach by building subVIs where you find a logical division of labor or the potential for code reuse.
- Solve your general problems along with your specific ones.
- Test your subVIs as you create them. You might need to construct higher-level test routines, but you can catch the bugs in one small module more easily than in a hierarchy of several VIs.

As you consider the details of your subVIs, you might find that your initial design is incomplete. For example, you might realize you need to transfer more information from one subVI to another. You might have to reevaluate your top-level design at this point. Using modular subVIs to accomplish specific tasks makes it easier to manage your program reorganizations.

Plan Ahead with Connector Panes

If you think that you might need to add additional inputs or outputs later on, select a connector-pane pattern with extra terminals. You can leave these extra terminals unconnected. With these extra terminals, you do not have to change the connector pane for your VI if you find you need another input or output later. This flexibility enables you to make these changes with minimal effect on your hierarchy.

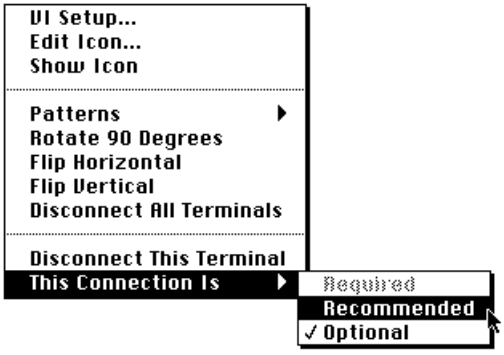
When linking controls and indicators to the connector, place inputs on the left and outputs on the right. This prevents complicated, unclear wiring patterns in your VIs.



If you create a group of subVIs that are used together often, try to give the subVIs a consistent connector pane, with common inputs in the same location. You then can remember where to locate each input more easily without using the Help window. If you create a subVI that produces an output that is used as the input to another subVI, try to align the input and output connections. This technique simplifies your wiring patterns.

SubVIs with Required Inputs

On the front panel, you can edit required inputs for subVIs by clicking the icon pane on the upper-right side of the window and choosing **Show Connector»This Connection Is**. From the submenu, choose between the **Required**, **Recommended**, or **Optional** options.
The following illustration displays the submenu options.



If you want to return to the icon pane in the front panel, pop up on the connector pane and select **Show Icon**.

Good Diagram Style

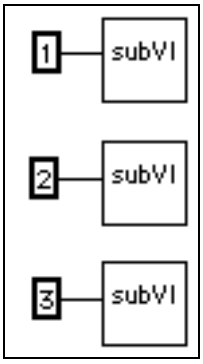
In general, avoid creating a block diagram that uses more than one or two screens of space. If a diagram becomes very large, decide whether you can reuse some components of your diagram in other VIs, or whether a section of your diagram fits together as a logical component. If so, consider dividing your diagram into subVIs.

With forethought and careful planning, it is easier to design diagrams that use subVIs to perform specific tasks. Using subVIs helps you manage changes and debug your diagrams quickly. You can determine the function of a well-structured program after only a brief examination.

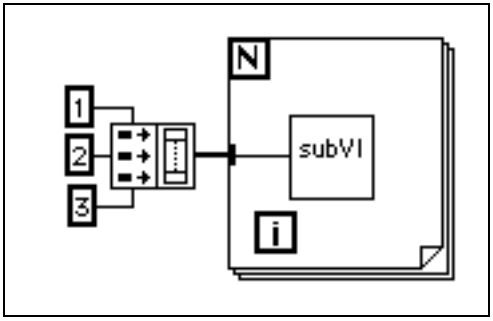
Watch for Common Operations

As you design your programs, you might find that you perform a certain operation frequently. Depending on the situation, consider using subVIs or loops to perform an action repetitively.

For example, examine the following diagram in which three similar operations run independently.



An alternative to this design is a loop, which performs the operation three times. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the loop.



If the array elements are constant, you can use an array constant instead of building the array on the block diagram.

Use Left-to-Right Layouts

G is designed to use a left-to-right (and sometimes top-to-bottom) layout. Organize all elements of your program in this layout when possible.

Check for Errors

When you perform any kind of I/O, consider the possibility of errors occurring. Almost all I/O functions return error information. If you use direct I/O, make sure that your program checks for errors and you handle them appropriately.

LabVIEW does not handle errors automatically because users usually want very specific error-handling methods. For example, if an I/O VI in your block diagram times out, you might or might not want your entire program to stop. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error-handling decisions in the block diagram of your VI.

The following list describes situations in which errors frequently occur:

- Incorrect initialization of communication or data that has been written to an external device improperly
- Loss of power in an external device, or a broken or improperly working external device
- Change in functionality of an application or library when upgrading operating system software

When an error occurs, you might not want certain subsequent operations to occur. For instance, if an analog output operation fails because you specify the wrong device, you might not want a subsequent analog input operation to take place.

One method for managing such a problem is to test for errors after every function and place subsequent functions inside case structures. However, this method can complicate your diagrams and ultimately hide the purpose of your application.

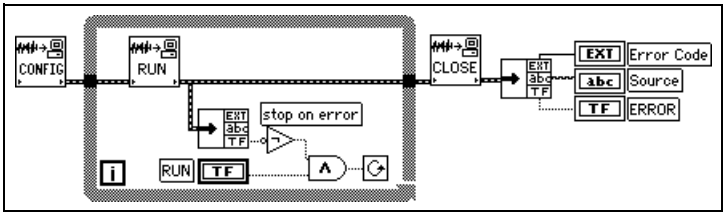
An alternative approach, which has been used successfully in a number of applications and many of the VI libraries, is to incorporate error handling in the subVIs that perform I/O. Each VI can have an error input and an error output. You can design the VI to check the error input to see if an error has occurred previously. If an error exists, you can configure the VI to do nothing and pass the error input to the error output. If no error exists, the VI can execute the operation and pass the result to the error output.

☞ **Note** ***In some cases, such as a Close operation, you might want the VI to perform the operation regardless of the error that is passed into it.***

Using the preceding technique, you can wire several VIs together, connecting error inputs and outputs to propagate errors from one VI to the next. At the end of the series of VIs, you can use the Simple Error Handler VI to display a dialog box if an error occurs. The Simple Error Handler VI is located in **Functions»Time & Dialog**. In addition to encapsulating error handling, you can use this technique to determine the order of several I/O operations.

One of the main advantages in using the error input and output clusters is that you can use them to control the execution order of dissimilar operations.

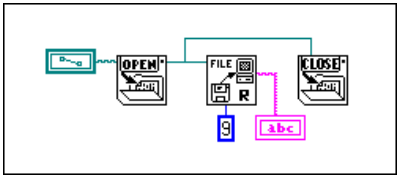
The error information generally is represented using a cluster containing a numeric error code, a string containing the name of the function that generated the error, and an error Boolean for quick testing. The following illustration shows how you can use this technique in your own applications. Notice that the While Loop stops if it detects an error.



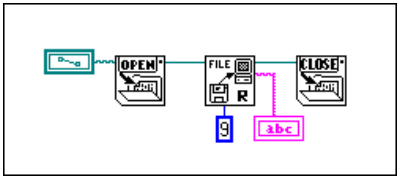
Watch Out for Missing Dependencies

Make sure that you have explicitly defined the sequence of events when necessary. Do not assume left-to-right or top-to-bottom execution when no data dependency exists.

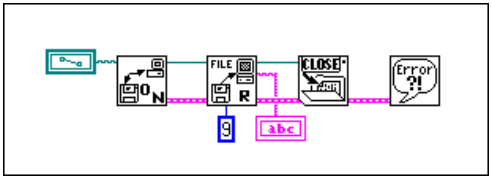
In the following example, no dependency exists between the Read File VI and the Close File VI. This program might not work as expected.



The following version of the block diagram establishes a dependency by wiring an output of the Read File VI to the Close File VI. The operation cannot end until the Close File VI receives the output of the Read File VI.



Notice that the preceding example still does not check for errors. For instance, if the file does not exist, the program does not display a warning. The following version of the block diagram illustrates one technique for handling this problem. In this example, the block diagram uses the error I/O inputs and outputs of these functions to propagate any errors to the Simple Error Handler VI.



Avoid Overuse of Sequence Structures

Because VIs can operate with a great deal of inherent parallelism, avoid using Sequence structures. Using a Sequence structure guarantees the order of execution but prohibits parallel operations. For instance, asynchronous tasks that use I/O devices (GPIB, serial ports, and data acquisition boards) can run concurrently with other operations if Sequence structures do not prevent them from doing so.

Sequence structures tend to hide parts of the program and interrupt the natural left-to-right flow of data. You do not sacrifice performance by using Sequence structures. However, when you need to sequence operations, you might consider using data flow instead. For instance, in I/O operations you might use the error I/O technique described previously to ensure that one I/O operation occurs before another.

Study the Examples

For further information about program design, you can examine the many example block diagrams included in LabVIEW. These sample programs provide you with insights into G programming style and technique.
To view these block diagrams, open any of the VIs in the `Examples` directory.

29

Where to Go from Here

You have completed activities that have prepared you to create LabVIEW applications. Before you start your own applications, you might want to examine the additional resources available to you.

Other Useful Resources

The following sections overview resources you can use to create your applications successfully.

Solution Wizard and Search Examples

To find examples similar to your application, open the **Solution Wizard** and the **Search Examples** options from the LabVIEW dialog box. The **Solution Wizard** creates data acquisition and instrument I/O examples based on the criteria you specify. The **Search Examples** option opens examples illustrating several G programming concepts as well as analysis, networking, data acquisition, and instrument I/O.

Data Acquisition Applications

(Windows, Macintosh) The *LabVIEW Data Acquisition Basics Manual* provides information about starting data acquisition applications that use analog input, analog output, digital I/O, and counter/timers. The *LabVIEW Data Acquisition Basics Manual* also explains the basic concepts behind data acquisition and the VIs used to implement those concepts.

G Programming Techniques

Part I, *Introduction to G Programming*, and Part V, *Advanced G Programming*, in this manual have introduced you to fundamental G programming techniques. To learn more about the capabilities of G, refer to the *G Programming Reference Manual*, which provides further information about execution and debugging, VI setup, front panel objects, wiring, structures, and attribute nodes. It also provides information not discussed in the *LabVIEW User Manual*, such as printing, customizing the G environment using **Preferences**, custom controls, multithreading, and performance issues.

Function and VI Reference

For an overview of the functions and VIs available in LabVIEW, refer to the *LabVIEW Function and VI Reference Manual*, which provides brief descriptions of the functions and VIs, organized in the same order in which they appear in the **Functions** palette.

Resources for Advanced Topics

The *LabVIEW User Manual* teaches you the fundamentals of building a LabVIEW application. LabVIEW contains several advanced features that are either not discussed or only discussed in a limited fashion in this manual. The following sections overview these features and provide additional resources so you can apply them as necessary in your applications.

Attribute Nodes

Chapter 27, *Front Panel Object Attributes*, of this manual briefly describes attribute nodes. Using attribute nodes, you can manage settings related to controls and indicators programmatically. For example, you can change the visibility of controls. Use an attribute node if you need to change the options in a ring or list control, clear the contents of a chart, or change the scales on a chart or graph programmatically. Chapter 22, *Attribute Nodes*, in the *G Programming Reference Manual* describes attribute nodes in detail.

VI Setup and Preferences

You can programmatically control various attributes of VIs as well as attributes of LabVIEW using the VI Server feature. You can load VIs into memory, run VIs, change the appearance of VIs, and change the execution behavior of a VI. Some of the options that you can set interactively in **Preferences** and **VI Setup…** you can also set programmatically. The options available in **Preferences** are considered application settings because they affect all VIs shown in LabVIEW. The **VI Setup…** options are VI settings because they only affect one VI and all instances in which you use the VI as a subVI.

In addition to changing attributes, which are called properties, you can perform an action on LabVIEW or an individual VI. For example, you can set an action, also called a method, to save a VI. You can set properties and methods for LabVIEW and VIs on the local machine, across a TCP/IP network, or through another ActiveX application. For more information

about ActiveX capabilities, refer to Chapter 22, *ActiveX Support*, in this manual. For more information about TCP/IP settings and how to use this feature, refer to Chapter 7, *Customizing Your G Environment*, and Chapter 21, *VI Server*, in the *G Programming Reference Manual*.

Local and Global Variables

Use local variables to read from controls in multiple locations of the block diagram. Also use local variables when you need to treat a front panel object as a control in some locations and an indicator in other locations. Use local variables sparingly because they hide the data flow of your diagrams, which makes it difficult to see the purpose of your program and to debug local variables. See Chapter 23, *Global and Local Variables*, of the *G Programming Reference Manual* for a discussion of local variables.

Global variables store data used by several VIs. Use global variables judiciously because they hide the data flow of your diagram. Although you need global variables in some applications, do not use them if you can structure your program using an alternate data flow method for transferring data. See Chapter 23, *Global and Local Variables*, of the *G Programming Reference Manual* for details.

Creating SubVIs

You can create subVIs from a selection on the block diagram by choosing **Edit»Create SubVI from Selection**. In addition, LabVIEW automatically wires the correct inputs and outputs to the subVI. In some instances, you cannot create a subVI from a VI. See Chapter 3, *Using SubVIs*, of the *G Programming Reference Manual* for a detailed discussion of this feature.

VI Profiles

You can use the VI profile feature (**Project»Show Profile Window**) to access detailed information about a timing statistics and timing details of a VI. This feature helps you optimize the performance of your VIs. See Chapter 28, *Performance Issues*, of the *G Programming Reference Manual* for a detailed discussion of the profile feature.

Control Editor

Use the Control Editor to customize the look of your front panel controls. You also can use the editor to save customized controls so you can reuse them in other applications. See Chapter 24, *Custom Controls and Type Definitions*, of the *G Programming Reference Manual* for a detailed discussion of the Control Editor.

List and Ring Controls

Use list and ring controls when you need to present the user with a list of options. See Chapter 13, *List and Ring Controls and Indicators*, in the *G Programming Reference Manual* for a detailed discussion of these front panel objects.

Call Library Function

LabVIEW provides a Call Library function you can use to call a shared library or DLL. With this function, you can create an interface in LabVIEW to call an existing code or driver. See Chapter 25, *Calling Code from Other Languages*, in the *G Programming Reference Manual* for a discussion of the Call Library functions.

Code Interface Nodes

You can use code interface nodes (CINs) as an alternative method for calling source code written in a conventional, text-based programming language from LabVIEW block diagrams. Use CINs for tasks that conventional programming languages can perform more quickly than LabVIEW, for tasks that you cannot perform directly from the block diagram, and for linking existing code to LabVIEW. However, the Call Library function generally is easier to use when calling source code than CINs. Use CINs when you need tighter integration with LabVIEW and the source code. See the *LabVIEW Code Interface Reference Manual*, available in Portable Document Format (PDF) only, and Chapter 25, *Calling Code from Other Languages*, in the *G Programming Reference Manual* and for further information about CINs.

A

Analysis References

This appendix lists the reference material used to produce the Analysis VIs in this manual. These references contain more information on the theories and algorithms implemented in the analysis library.

Baher, H. *Analog & Digital Signal Processing*. New York: John Wiley & Sons, 1990.

Bates, D.M. and Watts, D.G. *Nonlinear Regression Analysis and its Applications*. New York: John Wiley & Sons, 1988.

Bracewell, R.N. "Numerical Transforms." *Science* 248 (11 May 1990).

Burden, R.L. and Faires, J.D. *Numerical Analysis.* 3d ed. Boston: Prindle, Weber & Schmidt, 1985.

Chen, C.H. et al. *Signal Processing Handbook*. New York: Marcel Decker, Inc., 1988.

DeGroot, M. *Probability and Statistics*. 2d ed. Reading, Massachusetts: Addison-Wesley Publishing Co., 1986.

Dowdy, S. and Wearden, S. *Statistics for Research*. 2nd ed. New York: John Wiley & Sons. 1991.

Dudewicz, E.J. and Mishra, S.N. *Modern Mathematical Statistics.* New York: John Wiley & Sons, 1988.

Duhamel, P. et al. "On Computing the Inverse DFT." *IEEE Transactions on ASSP* 34, 1 (February 1986).

Dunn, O. and Clark, V. *Applied Statistics: Analysis of Variance and Regression* 2nd ed. New York: John Wiley & Sons. 1987.

Elliot, D.F. *Handbook of Digital Signal Processing Engineering Applications*. San Diego: Academic Press, 1987.

Golub, G.H. and Van Loan, C.F. *Matrix Computations.* Baltimore: The John Hopkins University Press, 1989.

Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” Proceedings of the IEEE-66 (1978)-1.

Maisel, J.E. “Hilbert Transform Works With Fourier Transforms to Dramatically Lower Sampling Rates.” *Personal Engineering and Instrumentation News* 7, 2 (February 1990).

Miller, I. and Freund, J.E. *Probability and Statistics for Engineers*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1987.

Neter, J. et al. *Applied Linear Regression Models.* Richard D. Irwin, Inc., 1983.

Neuvo, Y., Dong, C.-Y., and Mitra, S.K. “Interpolated Finite Impulse Response Filters” *IEEE Transactions on ASSP.* ASSP-32, 6 (June, 1984).

O’Neill, M.A. “Faster Than Fast Fourier.” *BYTE*. (April 1988).

Oppenheim, A.V. and Schafer, R.W. *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

Parks, T.W. and Burrus, C.S. *Digital Filter Design*. New York: John Wiley & Sons, Inc., 1987.

Pearson, C.E. *Numerical Methods in Engineering and Science*. New York: Van Nostrand Reinhold Co., 1986.

Press, W.H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1988.

Rabiner, L.R. & Gold, B. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1975.

Sorensen, H.V. et al. “On Computing the Split-Radix FFT.” *IEEE Transactions on ASSP.* ASSP-34, 1 (February 1986).

Sorensen, H.V. et al. “Real-Valued Fast Fourier Transform Algorithms.” *IEEE Transactions on ASSP.* ASSP-35, 6 (June 1987).

Spiegel, M. *Schaum’s Outline Series on Theory and Problems of Probability and Statistics.* New York: McGraw-Hill, 1975.

Stoer, J. and Bulirsch, R. *Introduction to Numerical Analysis*. New York: Springer-Verlag, 1987.

Vaidyanathan, P.P. *Multirate Systems and Filter Banks*. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

Wichman, B. and Hill, D. “Building a Random-Number Generator: A Pascal Routine for Very-Long-Cycle Random-Number Sequences.” *BYTE* (March 1987): 127–128.

B

Common Questions

This appendix answers common questions about LabVIEW networking communications and Instrument I/O, specifically GPIB and serial I/O.

Communications Common Questions

This section answers common questions about LabVIEW networking communications. Questions are divided into sections according to the relevant platform: All, Windows, and Macintosh. Please contact National Instruments if you have further questions or suggestions regarding LabVIEW.

Questions for All Platforms

How do I use LabVIEW to communicate with other applications?

Communicating with other applications, often called interprocess or interapplication communication, can be done with the standard networking protocols on each platform. LabVIEW has support for TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) on all platforms.

(Windows) In addition, LabVIEW for Windows supports DDE (Dynamic Data Exchange).

(Macintosh) In addition, LabVIEW for Macintosh supports IAC (Interapplication Communication). IAC includes Apple Events and PPC (Program to Program Communication).

(UNIX) LabVIEW for UNIX only supports TCP and UDP.

See Part II, *I/O Interfaces*, for more information on how use LabVIEW to communicate with other applications. In addition, for many instrumentation applications, file I/O provides a simple, adequate method of sending information between applications.

How do I launch another application with LabVIEW?

On Windows and UNIX, use `System Exec.vi` (**Functions»Communication**). On Macintosh, use AESend Finder Open (**Functions»Communication»AppleEvent**).

When would I want to use UDP instead of TCP?

Typically, UDP is used in applications where reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic. Also, UDP can be used to broadcast to any machine wanting to listen to the server.

What port numbers can I use with TCP and UDP?

A port is represented by a number between 0 and 65535. With UNIX, port numbers less than 1024 are reserved for privileged applications (for example, ftp). When you specify a local port, you can use the value 0 to use or to choose an unused port.

Why can't I broadcast using UDP?

Since the broadcast address varies among domains, you will need to find out from your system administrator what broadcast address to use. For example, the broadcast address 0xFFFFFFFF will not be correct for your domain. Additionally, your machine by default may not allow broadcasting unless the process is run by the root user.

Windows Only

Which WinSock DLLs can I use with LabVIEW?

This question pertains to Windows 3.*x* only because Windows 95 and Windows NT include this file.

Any WinSock driver that conforms to standard 1.1 should work with LabVIEW.

Recommended

National Instruments recommends using the WinSock DLL provided by Microsoft for Windows for Workgroups. Prior to the release of this WinSock DLL, National Instruments tested a number of WinSock DLLs. The following DLLs were recommended at that time.

- TCPOpen version 1.2.2 from Lanera Corporation.
- Trumpet version 1.0.
- Super-TCP version 3.0 R1 from Frontier Technologies Corporation.
- NEWT/Chameleon version 3.11 from NetManage, Inc.

Not Recommended

National Instruments' limited testing of these products yielded various problems and crashes while attempting TCP/IP communication. At this time we can neither recommend these products nor support customers attempting TCP/IP communication with these WinSock DLLs.

- Distinct TCP/IP version 3.1 from Distinct Corporation.
- PCTCP version 2.*x* from FTP Software, Inc.

How do I call an Excel macro using DDE?

Use `DDE Execute.vi`. This VI tells the DDE server to execute a command string in which you specify the action for Excel to perform and the name of the macro. Make sure to include the correct parentheses and brackets around the command. Refer to the *Excel User's Guide* for more information. Some common examples are shown below:

Command String	Action
[RUN("MACRO1")]	Runs MACRO1
[RUN("MACRO1!R1C1")]	Runs MACRO1 starting at Row 1, Column 1
[OPEN("C:\EXCEL\SURVEY.XLS")]	Opens `SURVEY.XLS`

Why doesn't DDE Poke work with Microsoft Access?

Microsoft Access cannot accept data directly from DDE clients. To get data into an Access database you must create a macro in that database to import the data from a file. In the simple case these macros need only be two actions long. First do a SetWarnings to suppress Access dialogs, then do a TransferSpreadsheet or TransferText to get the data. After this macro is defined, you can call the macro by sending an execute to that database with the macro name as the data. Refer to the example VI `Sending Data to Access.vi` located in `examples\network\access.llb` to see how this is done.

Which commands do I use to communicate with a non-LabVIEW application using DDE?

The DDE commands are specific to the application with which you are interfacing. Consult the documentation for the application to see which commands are available.

How do I install LabVIEW as a shared application on a file server?

If you have a license for each client, follow these procedures:

- Install the LabVIEW Full Development System on the server. (Unless there is NI hardware on the server, it is not necessary to install NI-DAQ or `GPIB.DLL`).
- Each local machine should use its own `labview.ini` file for LabVIEW preferences. If a `labview.ini` file does not already exist on a local machine, you can create this (empty) text document using a text editor, such as Microsoft Notepad. The first line of `labview.ini` must be [`labview`]. To have a local setting for `labview.ini`, LabVIEW requires a command line argument containing the path to the preferences. For example, if the `labview.exe` file is on drive `W:\LABVIEW` and the `labview.ini` file is on `C:\LVWORK` (the hard drive on the local machine), modify the command line option of the LabVIEW icon in Program Manager to be

  ```
  W:\LABVIEW\LABVIEW.EXE -pref C:\LVWORK\LABVIEW.INI
  ```

☞ **Note** `pref` ***must be lower case. Additionally, each local machine must have its own LabVIEW temporary directory, which you can specify in LabVIEW by choosing Edit»Preferences....***

- You do not need `GPIB.DLL` on the server machine, unless you are using a GPIB board on this machine. You will need the `gpibdrv` file in the LabVIEW directory. On each machine that has a GPIB board, you will need to install the driver for that board. Either use the drivers that came with the board or complete a custom LabVIEW installation in which you install only the desired GPIB driver on the local machine.
- The same procedure for `GPIB.DLL` applies to NI-DAQ.

On Windows NT why does the Synch DDE Client / Server hang after many transfers?

There are some problems with DDE in LabVIEW for Windows NT that result in VIs hanging during DDE Poke and DDE Request operations. This limitation is specific to Windows NT.

Macintosh Only

What is a target ID?

Target ID is used in the Apple Events and PPC VIs on the Macintosh. It serves as a reference to the application that you are trying to launch, run, or abort. You can obtain the target ID to an application using one of the following VIs:

- Get Target ID takes the name and location of the application as input, searches the network for it, and returns the target ID.
- PPC Browser pops up a dialog box that allows you to select an application, which may be across the network or on your computer.

You can use the target ID you obtain as an input to all subsequent Apple Event functions to open, print, close, or run the application.

Why can't I see my application in the dialog box generated by PPC Browser?

If the application you want to connect to cannot be used with Apple Events, it will not show up in the `PPC Browser` dialog box. If you are certain that the desired application supports Apple Events, open the **Apple** menu on the machine where the application is located. Select **Control Panels»File Sharing (MacOS 8)** or **Control Panels»Sharing Setup (System 7 and earlier)** and verify that **Program Linking** is turned on.

How can I close the Finder using Apple Events?

Use the VI `AESend Quit Application` to quit the Finder or any other application.

GPIB

All Platforms

When using a LabVIEW instrument driver, I have trouble talking with my instrument.

Ensure that your GPIB interface is working properly. Use the example `LabVIEW<->GPIB.vi` located in `examples\instr\smplgpib.llb`. Try a simple command to the instrument. For example, if the instrument is 488.2, the string `*IDN?` will ask the instrument to send its identification string (about 100 characters).

Once communication is established, you should have no problems with the instrument driver.

I get timeout errors with GPIB Read/Write in LabVIEW.

Try running a simple program to establish communication between LabVIEW and GPIB. Use the example `LabVIEW<->GPIB.vi`, located in `examples\instr\smplgpib.llb`. Try a simple command to the instrument. For example, if the instrument is 488.2, the string *IDN? will ask the instrument to send its identification string (about 100 characters).

If you still receive errors with GPIB, you may have a configuration problem. Open the GPIB configuration utility (Windows: `wibconf`; Mac: `NI-488 Config`; Sun: `ibconf`; HP-UX: `ibconf`). Verify that the settings match your hardware settings. Exit the configuration utility and run the `ibic` (Interface Bus Interactive Control) utility for your Windows: `wibic`; platform (Mac: `ibic`, which ships with your GPIB board; Sun: `ibic`; HP-UX: `ibic`).

Try the following sequence:

`: ibfind gpib0`	Find the GPIB interface
`id = 32000` `gpib0: ibsic`	Clear the GPIB bus (Send Interface Clear)
`[0130] [cmpl cic atn]`	Operation completed successfully
`gpib0: ibfind dev1`	Find device 1. Use appropriate instrument address.
`id = 32xxx` `dev1: ibwrt "*IDN?"`	Write string to instrument. 488.2 instruments recognize this command and return their identification string.
`count = 5` `dev1: ibrd 100`	Five bytes were sent. Read up to 100 bytes from the instrument.
`Fluke xxx Multimeter…`	Instrument returns identification string.

If you have any configuration errors, you will get an error message in one of these steps. The NI 488.2 Software Reference Manual that came with your GPIB board has detailed descriptions of the error messages.

Why can I communicate with my GPIB instrument with a LabVIEW VI running in execution highlighting mode but not when it is running full-speed?

This sounds like a timing problem. VIs run much slower with execution highlighting enabled than they will otherwise. Your instrument may need more time to prepare the data to send. Add a delay function or use service requests before the `GPIB Read.vi` to give the instrument enough time to generate the data it needs to send back to the computer.

Why can I write successfully to my GPIB instrument but can't read back from it?

When `GPIB Write.vi` executes, the computer is in talk mode and the instrument is in listen mode. `When GPIB Read.vi` executes, the device is supposed to switch to talk mode and the computer to listen mode.
The device is prompted to switch modes by a termination signal which may be a character (End Of String) or a GPIB bus line (End Or Identify). So, if `GPIB Read.vi` times out or returns an EABO (Operation aborted) error,

it means that the device is not receiving the right termination signal. To determine the termination mode for a given instrument, refer to its manual. As a rule of thumb, all IEEE 488.2 devices terminate on <CR><LF> and assertion of the EOI (End Or Identify) line in the GPIB bus.

Use the configuration utility for your platform (Windows: wibconf; Mac: NI-488 Config; Sun: ibconf; HP-UX: ibconf) to change the termination character.

A VI that talks with a GPIB instrument works fine on one platform and not on another.

Make sure that the instrument is configured properly in wibconf, NI-488 Config, or ibconf. Some older, 488.1 instruments do not automatically go to remote state. Go to the GPIB configuration utility for your platform and set the field Assert REN when SC. This will ensure that the instrument goes to remote state (as opposed to local state) when it is addressed.

Windows Only

I can communicate with my instrument using a Quick Basic program but not from LabVIEW.

The GPIB board has separate handlers for DOS and Windows. Quick Basic accesses the DOS handler, but LabVIEW accesses the Windows handler. Make sure that the board and device are configured properly through wibconf.exe.

Serial I/O

All Platforms

Why doesn't my instrument respond to commands sent with the Serial Port Write VI?

Many instruments expect a carriage return or line feed to terminate the command string. The Serial Port Write.vi in LabVIEW sends only those characters included in the string input; no termination character is appended to the string. Many terminal emulation (for example, Windows Terminal) packages automatically append a carriage return to the end of all transmissions. With LabVIEW, you will need to include the proper

termination character in your string input to `Serial Port Write.vi` if your instrument requires it.

Some instruments require a carriage return (`\r`); others require a line feed (`\n`). When you enter a return on the keyboard (on PC keyboards, this is the Enter key on the main alphanumeric keypad), LabVIEW inserts a `\n`. To insert a carriage return, use the Concatenate Strings function and append a carriage return constant to the string, or manually enter `\r` after selecting **'\' Codes Display** from the string pop-up menu.

Make sure that your cable works. Many of our technical support questions are related to bad cables. In computer to computer communication with serial I/O, use a null-modem to reverse the receive and transmit signals.

See the example `LabVIEW<->Serial.vi` to establish communication with your instrument. It is located in `examples\instr\smplserl.llb`. The VI also demonstrates the use of `Bytes at Serial Port.vi` before reading data back from the serial port.

How do I close a serial port so that other applications can use it?

You may wish to close the serial port after use. For example, on Windows, a VI may write information using `Serial Port Write.vi` to lpt1, connected to a printer. After the operation is complete, LabVIEW still has control over the serial port. Other applications cannot use this port until LabVIEW has released control.

LabVIEW contains a `Close Serial Driver.vi` on all platforms. This VI tells LabVIEW to release control over the specified port. The Close Serial Driver.vi is not in the **Serial** palette; it is located in `vi.lib\Instr\_sersup.llb`. To access the VI, use the **Functions**»**VI...** or **File**»**Open...** commands.

How do I clear my serial port buffer?

Read the remaining data in the buffer and ignore it.

How can I add additional serial ports to my computer?

You can add additional serial ports to your IBM-compatible PC using AT bus serial interface boards from National Instruments. If you are using another platform, you can use a third party board to add serial ports to your computer. Some third party boards require a special language interface that does not conform to the standard API for serial ports on the platform. In this case, you will need to write your own interface, probably through a CIN or DLL, to the driver. The following sections explain how to use the serial port

VIs included in LabVIEW to address boards which use the standard serial port interface on the different platforms.

(Windows 3.*x*) LabVIEW for Windows uses the standard Microsoft Windows interface to the serial port. Thus, limitations in the use of serial ports in LabVIEW are due to limitations in Windows. For example, because Windows can only address ports COM1 through COM9, LabVIEW can only address these ports. Further, Windows allows access to only eight serial ports at any time; thus, LabVIEW can control a maximum of eight serial ports at any one time.

National Instruments now sells Plug and Play AT serial boards for a variety of solutions for serial communications. The AT-485 and AT-232 asynchronous serial interface boards are available in either 2 or 4-port configurations. Full Plug and Play compatibility gives you the benefits of switchless configuration for easier installation and maintenance. The AT-485 and AT-232 include the following software components for use with Windows: device driver, hardware diagnostic test, and configuration utility. These boards have been tested with LabVIEW for Windows. Contact National Instruments for more information:

Manufacturer:	National Instruments
Product Name:	AT-485 and AT-232
Tel:	512 794 0100
Fax:	512 794 8411
Fax-on-Demand:	800 329 7177, order number 1430

A third party board which uses some work-around to overcome Windows limitations will in all probability not work well, if at all, with LabVIEW. It is possible to write a CIN or DLL to access such boards, but this is not recommended. In general, boards that use the standard Windows interface should be completely compatible with LabVIEW.

(Windows 95 and Windows NT) Unlike Windows 3.*x*, Windows 95 and Windows NT are not limited to eight serial ports. Under Windows 95 and Windows NT, LabVIEW can address as many as 256 serial ports. The default port number parameter is 0 for COM1, 1 for COM2, 2 for COM3 and so on.

The file `labview.ini` contains the LabVIEW configuration options. To set the devices which will be used by the serial port VIs, set the configuration option labview.serialDevices to the list of devices to be used. For example, to set up your devices in a way similar to Windows 3.*x*:

```
serialDevices="COM1;COM2;COM3;COM4;COM5;COM6;COM7;COM8;
COM9;\\.\COM10;\\.\COM11;\\.\COM12;\\.\COM13;\\.\COM14;
\\.\COM15;\\.\COM16;LPT1;LPT2"
```

The above should appear as a single line in your configuration.

(Macintosh) LabVIEW uses standard system INITs to talk with the serial ports. By default, LabVIEW uses the drivers `.aIn` and `.aOut`, the modem port, as port 0, and drivers `.bIn` and `.bOut`, the printer port, for port 1. To access additional ports, you must install additional boards with the accompanying INITs.

After the board and INIT(s) are installed, tell LabVIEW how to use the additional ports. The method used in LabVIEW 5.0 is slightly different to that used in previous versions.

Modify the global `serpOpen.vi` (located in `vi.lib\Instr\_sersup.llb`) to accommodate the additional ports. All serial port VIs call `Open Serial Driver.vi`, which reads the appropriate input and/or output driver names from the `serpOpen.vi`. The front panel of `serpOpen.vi` contains two string arrays, named input driver names and out driver names.

When `Open Serial Driver.vi` is called, it uses the port number input as the index into the string arrays. Therefore, to allow LabVIEW to recognize additional serial ports, add additional string elements into the input and output driver names, then select **Make Current Values Default** and save the changes to `serpOpen.vi`.

Each serial port on a plug-in board has two names, one for input and one for output. The exact names and instructions for installing the drivers comes with the documentation for the board. Contact the board manufacturer if the instructions are missing or unclear.

The following boards work with LabVIEW for Macintosh:

Manufacturer:	Creative Solutions, Inc.
Product Name:	Hurdler HQS (4 ports) or HDS (2 ports)
Phone:	301 984 0262
Fax:	301 770 1675

Manufacturer:	Greensprings
Product Name:	RM 1280 (4 ports)
Phone:	415 327 1200
Fax:	415 327 3808

(UNIX) On a Sun SPARCstation under Solaris 1 and on Concurrent PowerMAX, the **port number** parameter for the serial port VIs is 0 for `/dev/ttya`, 1 for `/dev/ttyb`, and so on. Under Solaris 2, port 0 refers to `/dev/cua/a`, 1 to `/dev/cua/b`, and so on. Under HP-UX port number 0 refers to `/dev/tty00`, 1 to `/dev/tty01`, and so on.

On Concurrent PowerMAX, port 0 refers to `/dev/console`, Port 1 refers to `/dev/tty1`, Port 2 refers to `/dev/tty2`, and so on.

Because other vendor's serial port boards can have arbitrary device names, LabVIEW has developed an easy interface to keep the numbering of ports simple. In LabVIEW for Sun, HP-UX, and Concurrent PowerMAX, a configuration option exists to tell LabVIEW how to address the serial ports. LabVIEW supports any board that uses standard UNIX devices. Some manufacturers suggest using `cua` rather than `tty` device nodes with their boards. LabVIEW can address both types of nodes.

The file `.labviewrc` contains the LabVIEW configuration options. To set the devices the serial port VIs use, set the configuration option `labview.serialDevices` to the list of devices you intend to use.

For example, the default is:

```
labview.serialDevices:/dev/ttya:/dev/ttyb:/dev/ttyc:...
:/dev/ttyz.
```

☞ **Note** ***This requires that any third party serial board installation include a method of creating a*** `standard/dev file` ***(node) and that the user knows the name of that file.***

The following boards should work with LabVIEW for Sun:

Manufacturer:	Sun
Product Name:	SBus Serial Parallel Controller (8 serial, 1 parallel)
Manufacturer:	Artecon, Inc.
Product Name:	SUNX-SB-300P ArtePort SBus Card with 3 Ser / 1 Par Ports SUNX-SB-400P ArtePort SBus Card with 4 Ser / 1 Par Ports SUNX-SB-1600 ArtePort SBus Card with 16 Serial Ports

For any of these products contact SunExpress at (800) 873-7869.

How can I control the DTR and RTS serial lines?

`Serial Port Init.vi` can be used to configure the serial port for hardware handshaking; however, some applications may require manual toggling of the DTR and RTS lines. Because the interface to the serial ports is platform-dependent, each platform has a separate mechanism to control the lines.

(Windows) The LabVIEW for Windows distribution contains a VI which you can use to drive the DTR and RTS serial lines. The VI `serial line ctrl.vi`, located in `vi.lib\Instr\_sersup.llb`, can be used to control these lines. The VI will toggle these lines according to the function input. Valid codes for the input are:

0 noop
1 clear DTR
2 set DTR
3 clear RTS
4 set RTS
5 set DTR protocol
6 clr DTR protocol
7 noop2

(Macintosh) On the Macintosh, you can use the Device Control/Status function to control the serial port. *Inside Macintosh* (see Volume II, pages 245–259 and Volume IV, pages 225–228), contains specific information on what csCodes can be sent to the serial port. A summary of the codes and their functions is listed here:

code	param	Effect
13	baudRate	Set baud rate (actual rate, as an integer)
14	serShk	Set handshake parameters
16	byte	Set miscellaneous control options
17		Asserts DTR
18		Negates DTR
19	char	Replace parity errors
20	2 chars	Replace parity errors with alternate character
21		Unconditionally set XOff for output flow control
22		Unconditionally clear XOff for output flow control
23		Send XOn for input flow control if XOff was sent last
24		Unconditionally send XOn for input flow control
25		Send XOff for input flow control if XOn was sent last
26		Unconditionally send XOff for input flow control
27		Reset SCC channel

(Sun) LabVIEW for Sun contains no specific support to toggle the hardware handshaking lines of the serial ports. To manually control these lines, you must write a CIN. See the *Steps for Creating a CIN* section in Chapter 1, *CIN Overview*, of the *LabVIEW Code Interface Reference Manual*, available in Portable Document Format (PDF) only, for further details on how to write a CIN.

Why can't I allocate a serial buffer larger than 32kbytes?

You can not use a buffer size on `Serial Port Init.vi` that is larger than 32k because Windows and Macintosh limit the serial port buffer to 32k; thus, if you allocate a buffer larger than this, LabVIEW truncates the buffer size to 32k. This is not a problem on the Sun.

Windows Only

How do I access the parallel port?

In LabVIEW for Windows 3.*x*, port 10 is LPT1, port 11 is LPT2, and so on. For Windows 95/NT, you can set a port to be LPT1 in Serial Devices. To send data to a printer connected to a parallel port, use the `Serial Port Write.vi`.

What do the error numbers received from the serial port VIs mean?

The Serial Port VIs in LabVIEW for Windows return the errors reported by the Windows GetCommError function. Error numbers returned by the Serial Port VIs are 0x4000 (16,384) 'Or'-ed with the error numbers in the following table. Notice that the error returned reflects the status of the serial port; the error may have been generated as the result of a previous serial port function. The return values can be a combination of the following errors:

Hex Value	Error Name	Heading
0x0001	CE_RXOVER	Receiving queue overflowed. There was either no room in the input queue or a character was received after the end-of-file character was received.
0x0002	CE_OVERRUN	Character was not read from the hardware before the next character arrived. The character was lost.
0x0004	CE_RXPARITY	Hardware detected a parity error.
0x0008	CE_FRAME	Hardware detected a framing error.
0x0010	CE_BREAK	Hardware detected a break condition.

Hex Value	Error Name	Heading
0x0020	CE_CTSTO	CTS (clear-to-send) timeout. While a character was being transmitted, CTS was low for the duration specified by the fCtsHold member of COMSTAT.
0x0040	CE_DSRTO	DSR (data-set-ready) timeout. While a character was being transmitted, DSR was low for the duration specified by the fDsrHold member of COMSTAT.
0x0080	CE_RLSDTO	RLSD (receive-line-signal-detect) timeout.While a character was being transmitted, RLSD was low for the duration specified by the fRlsdHold member of COMSTAT.
0x0100	CE_TXFULL	Transmission queue was full when a function attempted to queue a character.
0x0200	CE_PTO	Timeout occurred during an attempt to communicate with a parallel device.
0x0400	CE_IOE	I/O error occurred during an attempt to communicate with a parallel device.

Hex Value	Error Name	Heading
0x0800	CE_DNS	Parallel device was not selected.
0x1000	CE_OOP	Parallel device signaled that it is out of paper.
0x8000	CE_MODE	Requested mode is not supported, or the `idComDev` parameter is invalid. If set, CE_MODE is the only valid error.

To use this table, take the error number and dissect it into its error components. For example, if `Serial Port Write.vi` returns the error 16,408, then the errors returned are `CE_BREAK` and `CE_FRAME` (16,408 = 16,384 + 16 + 8 = 0x4000 + 0x0010 + 0x0008).

Sun Only

I receive an error –37 when performing serial I/O.

Error –37 means that LabVIEW cannot find the appropriate serial device. This indicates that either a) the `/dev/tty?` files do not exist on your machine, or b) LabVIEW cannot find the file serpdrv.

By default, LabVIEW addresses `/dev/ttya` as port 0, `/dev/ttyb` as port 1, and so on. These devices must exist and the user must have read and write permissions to access the devices. You can change the devices LabVIEW accesses with the serial port VIs by adding the serialDevices configuration option to your `.Xdefaults` file. See the *LabVIEW Release Notes* on how to use this option.

The `serpdrv` file is shipped with LabVIEW and serves as the interface between LabVIEW and the Sun serial ports. This file needs to be in the location specified by the libdir configuration option, set to the LabVIEW directory by default. This means that serpdrv needs to be in the same directory as `gpibdrv` and `vi.lib`.

Serial I/O hangs on a Solaris 1.*x* machine.

LabVIEW for Sun uses asynchronous I/O calls when performing serial port operations. In the `Generic_Small` kernel, asynchronous I/O has been commented out. To access the serial ports from LabVIEW for Sun, the user must use the standard Generic kernel (not `Generic_Small`), or rebuild the `Generic_Small` kernel and reboot the SPARC.

C

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed_____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

LabVIEW Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *LabVIEW™ User Manual*

Edition Date: January 1998

Part Number: 320999B-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meanings	Value
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}

Numbers/Symbols

∞ Infinity.

π Pi.

D Delta. Difference. Δx denotes the value by which x changes from one index to the next.

1D One-dimensional.

2D Two-dimensional.

A

A/D Analog/digital.

absolute path Relative file or directory path that describes a location relative to the top level of the file system.

active window Window that is set to accept user input, usually the front-most window. The title bar of an active window is highlighted. Make a window active by clicking in it or by selecting it from the **Windows** menu.

ANSI American National Standards Institute.

array Ordered, indexed set of data elements of the same type.

array shell Front panel object that houses an array. An array shell 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 the execution of a node.
ASCII	American Standard Code for Information Interchange.
asynchronous execution	Mode in which multiple processes share processor time. For example, one process executes while others wait for interrupts during device I/O or while waiting for a clock tick.
ATE	Automatic test equipment.
auto-indexing	Capability of loop structures to disassemble and assemble arrays at their borders. As an array enters a loop with auto-indexing enabled, the loop automatically disassembles it with scalars extracted from one-dimensional arrays, one-dimensional arrays extracted from two-dimensional arrays, and so on. Loops assemble data into arrays as they exit the loop according to the reverse of the same procedure.
autoscaling	Ability of scales to adjust to the range of plotted values. On graph scales, this feature also determines maximum and minimum scale values.
autosizing	Automatic resizing of labels to accommodate text that you type.

B

block diagram	Pictorial description or representation of a program or algorithm. In G, the block diagram is the source code for the VI. It consists of executable icons called nodes as well as wires that carry data between the nodes.
BNF	Backus-Naur form. A common representation for language grammars in computer science.
Boolean controls and indicators	Front panel objects used to manipulate and display Boolean (TRUE or FALSE) data. Several styles are available, such as switches, buttons, and LEDs.
breakpoint	Point at which execution halts when a subVI is called. You set a breakpoint by clicking 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 run button	Button that replaces the Run button when a VI cannot run because of errors. Click this button to invoke the Error List dialog box.

broken VI — VI that cannot be compiled or run. It is signified by a broken arrow in the Run button.

Bundle node — Function that creates clusters from various types of elements.

byte stream file — File that stores data as a sequence of ASCII characters or bytes.

C

case — One subdiagram of a Case structure.

Case structure — Conditional branching control structure, which executes one and only one of its subdiagrams based on its input. It is the combination of the IF, THEN, ELSE, and CASE statements in control flow languages.

cast — To change the type descriptor of a data element without altering the memory image of the data.

chart — *See* scope chart, strip chart, and sweep chart.

check box — Small square box in a dialog box that can be selected or cleared. Check boxes generally are associated with multiple options that can be set. More than one check box can be selected.

CIN — *See* code interface node.

cloning — To make a copy of a control or other G object by clicking it, pressing <Ctrl> (Windows), <option> (Macintosh), <meta> (Sun), or <Alt> (HP-UNIX), and dragging the copy to a new location.

cluster — Set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.

cluster shell — Front panel object that contains the elements of a cluster.

code interface node — CIN. Special block diagram node through which you can link conventional, text-based code to a VI.

coercion — Automatic conversion G performs to change the numeric representation of a data element.

coercion dot — Gray dot on a terminal indicating that one of two terminals wired together has been converted to match the data type of the other.

Color Copy tool	Tool you use to copy colors for pasting with the Color tool.
Color tool	Tool you use to set foreground and background colors.
compile	Process that converts high-level code to machine-executable code. VIs are compiled automatically before they run for the first time after creation or alteration.
conditional terminal	Terminal of a While Loop that contains a Boolean value that determines whether the VI performs another iteration.
connector	Part of the VI or function node that contains input and output terminals. Data passes to and from the node through a connector.
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.
continuous run	Execution mode in which a VI is run repeatedly until the operator stops it. You enable it by clicking the **Continuous Run** button.
Continuous Run button	Icon that indicates the execution status of a VI.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically.
control flow	Programming system in which the sequential order of instructions determines execution order. Most conventional text-based programming languages, such as C, Pascal, and BASIC, are control flow languages.
Controls palette	Palette containing front panel controls and indicators.
conversion	Process of changing the type of a data element.
count terminal	Terminal of a For Loop whose value determines the number of times a For Loop executes its subdiagram.
CPU	Central processing unit.

current VI	VI whose front panel, block diagram, or Icon Editor is the active window.
custom PICT controls and indicators	Controls and indicators whose parts can be replaced by graphics and indicators you supply.

D

DAQ	*See* data acquisition.
data acquisition	DAQ. Process of acquiring data, typically from A/D or digital input plug-in boards.
data dependency	Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *See also* artificial data dependency.
data flow	Programming system consisting of executable nodes in which nodes execute only when they have received all required input data and produce output automatically when they have executed. G is a dataflow system.
data logging	To acquire data and simultaneously store it in a disk file. G file I/O functions can log data.
data storage formats	Arrangement and representation of data stored in memory.
data type	Format for information. In BridgeVIEW, acceptable data types for tag configuration are analog, discrete, bit array, and string. In LabVIEW, acceptable data types for most functions are numeric, array, string, and cluster.
data type descriptor	Code that identifies data types, used in data storage and representation.
datalog file	File that stores data as a sequence of records of a single, arbitrary data type that you specify when you create the file. While all the records in a datalog file must be of a single type, that type can be complex; for instance, you can specify that each record is a cluster containing a string, a number, and an array.
DDE	*See* dynamic data exchange.
description box	Dialog box containing online documentation for a G object.
destination terminal	*See* sink terminal.

developer	*See* system developer.
dialog box	Window that appears when an application needs further information to carry out a command.
dimension	Size and structure attribute of an array.
directory	Structure for organizing files into convenient groups. A directory is like an address showing where files are located. A directory can contain files or subdirectories of files.
drive	Letter in the range a–z, followed by a colon (:), indicating a logical disk drive.
DUT	Device under test.
dynamic data exchange	DDE. Passage of data between applications, accomplished without user involvement or monitoring.

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.
end of file	EOF. 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 message	Indication of a software or hardware malfunction, or an unacceptable data entry attempt.
execution highlighting	Feature that animates VI execution to illustrate the data flow in the VI.
external routine	*See* shared external routine.

F

FFT	Fast Fourier transform.
file refnum	Identifier that G associates with a file when you open it. You use the file refnum to indicate that you want a function or VI to perform an operation on the open file.

flattened data	Data of any type that has been converted to a string, usually, for writing it to a file.
For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to conventional code: `For i = 0 to n – 1, do....`
formula node	Node that executes formulas you enter as text. Formula nodes are 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 a front panel or block diagram that does not belong to any object.
front panel	Interactive user interface of a VI. Modeled after the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, or other controls or 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	Graphical programming language used to develop LabVIEW and BridgeVIEW applications.
global variable	Non-reentrant subVI with local memory that uses an uninitialized shift register to store data from one execution to the next. The memory of copies of these subVIs is shared and thus can be used to pass global data between them.
glyph	Small picture or icon.
GPIB	General Purpose Interface Bus. Common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.
graph control	Front panel object that displays data in a Cartesian plane.

H

handle	Pointer to a pointer to a block of memory that manages reference arrays and strings. An array of strings is a handle to a block of memory containing handles to strings.
Help	Online instructions that explain how to use a Windows application. The **Help** menu displays specific Help topics. Pressing <F1> displays a list of Help topics.
Help window	Special window that displays the names and locations of the terminals for a function or subVI, the description of controls and indicators, the values of universal constants, and descriptions and data types of control attributes.
hex	Hexadecimal. Base-16 number system.
hierarchical palette	Menu that contains palettes and subpalettes.
Hierarchy window	Window that graphically displays the hierarchy of VIs and subVIs.
housing	Nonmoving part of front panel controls and indicators that contains sliders and scales.
Hz	Hertz. Cycles per second.

I

I/O	Input/output. Transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.
icon	Graphical representation of a node on a block diagram.
Icon Editor	Interface similar to that of a paint program for creating VI icons.
icon pane	Region in the upper right corner of the Panel and Diagram windows that displays the VI icon.
IEEE	Institute for Electrical and Electronic Engineers.
indicator	Front panel object that displays output.
Inf	Digital display value for a floating-point representation of infinity.

inplace execution	Ability of a function or VI to reuse memory instead of allocating more.
instrument driver	VI that controls a programmable instrument.
iteration terminal	Terminal of a For Loop or While Loop that contains the current number of completed iterations.

L

label	Text object used to name or describe other objects or regions on the front panel or block diagram.
Labeling tool	Tool used to create labels and enter text into text windows.
LabVIEW	Laboratory Virtual Instrument Engineering Workbench. Program development application based on the programming language G used commonly for test and measurement purposes.
LED	Light-emitting diode.
legend	Object owned by a chart or graph that displays the names and styles of plots on that chart or graph.
list box	Box within a dialog box listing all available choices for a command. For example, a list of file names on a disk. Usually you select an item from the list box, then click **OK**. If more choices exist than can fit in the list box, it has vertical scroll bars. Selecting the down arrow next to the first item in the list displays the rest of the list box.
local variable	Variable that enables you to read or write to one of the controls or indicators on the front panel of your VI.

M

marquee	Moving, dashed border that surrounds selected objects.
matrix	Two-dimensional array.
MB	Megabytes of memory.

menu bar	Horizontal bar that lists the names of the main menus of an application. The menu bar appears below the title bar of a window. Each application has a menu bar that is distinct for that application, although some menus (and commands) are common to many applications.
mnemonic	String associated with an integer value.
modular programming	Type of programming that uses interchangeable computer routines.

N

NaN	Digital display value for a floating-point representation of *not a number* that typically is the result of an undefined operation, such as log(–1).
node	Program execution element. Nodes are analogous to statements, operators, functions, and subroutines in conventional programming languages. In a block diagram, nodes include functions, structures, and subVIs.
nondisplayable characters	ASCII characters that cannot be displayed, such as new line, tab, and so on.
not-a-path	Predefined value for the path control that indicates the path is invalid.
not-a-refnum	Predefined value that indicates the refnum is invalid.
numeric controls and indicators	Front panel objects used to manipulate and display or input and output numeric data.

O

object	Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures.
Object Pop-up Menu tool	Tool used to access a pop-up menu for an object.
OPC Server	OLE for Process Control. COM-based standard defined by the OPC foundation that specifies how to interact with device servers. COM is a 32-bit Windows technology.
Operating tool	Tool used to enter data into controls as well as operate them. Resembles a hand with a pointing finger.

P

palette	Display of icons that represent possible options.
pixmap	Standard format for storing pictures in which each pixel is represented by a color value. A bitmap is a black and white version of a pixmap.
platform	Computer and its operating system.
plot	Graphical representation of an array of data shown either in a graph or a chart.
polymorphism	Ability of a node to adjust automatically to data of different representation, type, or structure.
pop up	To call a special menu by right-clicking (Windows) or command-clicking (Macintosh) an object.
pop-up menu	Menu accessed by right-clicking (Windows) or command-clicking (Macintosh) an object. Menu options pertain to that object.
Positioning tool	Tool used to move, select, and resize objects. It resembles an arrow.
probe	Debugging feature for checking intermediate values in a VI.
Probe tool	Tool used to create probes on wires.
programmatic printing	Automatic printing of a VI front panel after execution.
pseudocode	Simplified language-independent representation of programming code.
pull-down menu	Menu accessed from a menu bar. Pull-down menu options usually are general in nature.

R

real-time	Pertaining to the performance of a computation during the actual time that the related physical process transpires so results of the computation can be used in guiding the physical process.
reentrant execution	Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.

refnum	Identifier of a DDE conversation or open file that can be referenced by related VIs.
representation	Subtype of the numeric data type. Representations include signed and unsigned byte, word, and long integers, as well as single-, double-, and extended-precision floating-point numbers, both real and complex.
resizing handle	Angled handle on the corner of an object that indicates a resizing point.
ring control	Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

S

scalar	Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans and clusters are explicitly singular instances of their respective data types.
scale	Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
scope chart	Numeric indicator modeled on the operation of an oscilloscope.
Scroll tool	Tool used to move through windows.
sensor	Device that produces a voltage or current output representative of a physical property being measured, such as speed, temperature, or flow.
sequence local	Terminal that passes data between the frames of a Sequence structure.
Sequence structure	Program control structure that executes its subdiagrams in numeric order. It commonly is used to force nodes that are not data-dependent to execute in a desired order.
shared external routine	Subroutine that can be shared by several CIN code resources.
shift register	Optional mechanism in loop structures used to pass the value of a variable from one iteration of a loop to a subsequent iteration.
sink terminal	Terminal that absorbs data. Also called a destination terminal.
slider	Moveable part of slide controls and indicators.

Solutions Gallery	Option within the DAQ Solution Wizard in which you can select from numerous categories of common DAQ applications.
source terminal	Terminal that emits data.
string controls and indicators	Front panel objects used to manipulate and display or input and output text.
strip chart	Numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.
structure	Program control element, such as a Sequence, Case, For Loop, or While Loop.
stubVI	Nonfunctional prototype of a subVI. A stubVI has inputs and outputs, but is incomplete. It is used during early planning stages of a VI design as a place holder for future VI development.
subdiagram	Block diagram within the border of a structure.
subpalette	Palette contained in an icon of another palette.
subVI	VI used in the block diagram of another VI. It is comparable to a subroutine.
sweep chart	Numeric indicator modeled on the operation of an oscilloscope. It is similar to a scope chart, except that a line sweeps across the display to separate old data from new data.
system developer	Creator of the application software to be executed.

T

table-driven execution	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 numbers.
terminal	Object or region on a node through which data passes.
tip strip	Text banner that displays the name of an object, control, or terminal.
tool	Special cursor you can use to perform specific operations.
toolbar	Bar containing command buttons you can use to run and debug VIs.

Tools palette	Palette containing the tools you use to edit and debug front-panel and block-diagram objects.
top-level VI	VI at the top of the VI hierarchy. This term distinguishes the VI from its subVIs.
tunnel	Data entry or exit terminal on a structure.
two-dimensional	Having two dimensions, such as an array with both rows and columns.
type descriptor	*See* data type descriptor.

U

universal constant	Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, such as pi.
user	*See* operator.
user-defined constant	Block diagram object that emits a value you set.
UUT	Unit under test.

V

VI	*See* virtual instrument.
VI library	Special file that contains a collection of related VIs for a specific use.
virtual instrument	VI. Program in the graphical programming language G; so-called because it models the appearance and function of a physical instrument.
Virtual Instrument Software Architecture	Single interface library for controlling GPIB, VXI, RS-232, and other types of instruments.
VISA	*See* Virtual Instrument Software Architecture.
VXI	VME eXtensions for Instrumentation (bus).

W

waveform chart	Indicator that plots data points at a certain rate.
While Loop	Loop structure that repeats a section of code until a condition is met. It is comparable to a Do loop or a Repeat-Until loop in conventional programming languages.
wire	Data path between nodes. *See also* data flow.
wire branch	Section of wire that contains all the wire segments from one junction to another, from a terminal to the next junction, or from one terminal to another if no junctions exist between the terminals.
wire junction	Point where three or more wire segments join.
wire segment	Single, horizontal or vertical piece of wire.
Wiring tool	Tool used to define data paths between source and sink terminals.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z