Contents

Help Topics are available in the following categories:

LabVIEW Overview
Getting Started with LabVIEW
VI Library Overview
How to Find Examples

LabVIEW Reference
Front Panel Reference
Block Diagram Reference
Function and VI Reference
Advanced Topics
Error Codes

More...
Upgrading to 4.x
Common Questions
Support and Training
Related Products

To learn how to use Help, press F1

Error Codes

This topic contains a collection of the error codes returned by the LabVIEW system.

Analysis Error Codes -20001 to -20065
Data Acquistion VI Error Codes -10001 to -10920
Daq Configuration Utility Error Codes -60 to -197
TCP and UDP Error Codes 53 to 66
DDE Error Codes 14001 to 14020
PPC Error Codes -900 to -932
LabVIEW Specific PPC Error Codes 1 to 5
Apple Event Error Codes -1700 to -1719
LabVIEW Specific Error Codes for Apple Events 1000 to 1004
LabVIEW Function Error Codes 0 to 85
GPIB Error Codes 0 to 32
Instrument Driver Error Codes -1200 to -13xx
Serial Port Error Codes 61 to 65
VISA Error Codes -10738077360 to -1073807231

How To Use Help

Welcome to an overview of the various forms of help for LabVIEW available to you. This hypertext help file is only one of the forms of help available to you. LabVIEW also comes with its own built-in help mechanisms, a comprehensive set of paper manuals, and two online manuals in Adobe Acrobat PDF format. Some information appears in several forms, some information only appears in printed manuals.

For instance, the LabVIEW Tutorial Manual, the LabVIEW Data Acquisition (DAQ) Basics Manual, the tutorial portion of the Code Interface Reference Manual, and the LabVIEW VXI VI Reference Manual do not appear online. The tutorial is a set of lessons that guide you through the process of becoming familiar with the common operations you use in LabVIEW to create programs, or as we call them in LabVIEW, virtual instruments (VIs). You refer to the manual while actually trying out the exercises on your computer. The DAQ basics manual contains information to get your analog I/O, digital I/O and counter applications started. The CIRM tutorial guides you in the creation of Code Interface Nodes, while the VXI VI reference manual explains the VIs that communicate with National Instruments VXI hardware.

LabVIEW comes with an array of ready-made VIs for special applications. The data acquisition, and instrument I/O communicate with National Instruments hardware. LabVIEW also comes with analysis, communication, and utility VIs. All of these VIs and their parameters are documented in LabVIEW's help window and this help file as well as in manual form.

The LabVIEW VXI VI Reference Manual and the LabVIEW Code Interface Reference Manual come as Adobe Acrobat PDF files and can be viewed online after you install the Acrobat reader that comes with those files.

See the topics below for more information:

Using Adobe Acrobat

Using LabVIEW's built-in help
Getting Help
Online Help for Constants, Functions, and SubVI Nodes

Using this Hypertext Help file and Creating your own
Hints for Using this Help File
Creating Your Own Help Files

Getting Started/More help
Getting Started with LabVIEW
Contact NI

If you are upgrading to 4.0

If you have VIs that were created with a previous version of LabVIEW, they must be converted to LabVIEW 4.0 format.

Note: You should back up your files before converting them. Once the files are saved with LabVIEW 4.0, you will not be able to open them with previous versions.

The upgrade process is described in the LabVIEW upgrade notes. The upgrade notes also detail the differences between this version and previous versions. You should have received a copy of the upgrade notes if you received LabVIEW as part of an upgrade. If you received LabVIEW as part of a new order, you may not have a printed copy of these notes. These notes are also avaiable in electronic form in your LabVIEW directory along with the release notes.

You cannot directly upgrade VIs that were saved with versions of LabVIEW older than 3.0. If You want to do this you should contact National Instruments.

Conversion Process

The conversion part of upgrading is a fairly automatic process. When you open a VI that was saved with a previous version, LabVIEW will convert it to the new version.

Conversion can be very memory intensive. The conversion process is performed in memory, and the VIs on disk are not changed unless you choose to save the VIs. When LabVIEW loads the VI, and it finds that the VI calls subVIs that need to be converted, LabVIEW will convert those VIs in memory as well. This memory usage is much greater than what LabVIEW normally requires to load a set of VIs, since LabVIEW usually loads only the components it needs.

If you have a large number of VIs, you should convert your VIs in stages. Convert and save VIs that are lower in your VI hierarchy before loading the higher level VIs. This will minimize the amount of memory required.

See the Upgrade Notes for more details.

Common Questions

This topic contains commonly asked questions about LabVIEW. If you are having problems or are unsure about a LabVIEW concept, the answer or explanation may be contain herein.

DAQ
GPIB
Serial I/O
Charts and Graphs
Error Messages and Crashes
Miscellaneous
Platform Issues and Compatibilities
Printing
Communication

Contact NI

Getting Started with LabVIEW

The topics in this section introduce the basics of the LabVIEW approach to programming. You may want to read the *LabVIEW Release Notes* and the *LabVIEW Tutorial Manual* before you read these topics.

Introduction to LabVIEW
Creating VIs
Editing VIs
Creating SubVIs
Finding VIs, Objects, and Text
- Using the Find Dialog Box
- Using Find Options from a Pop-Up Menu

Operating VIs
Debugging VIs
Creating Pop-Up Panels and Setting Window Features
Printing
- Printing in LabVIEW
- Using the Print Documentation Menu Option
- Programmatic Printing
- Other Methods for Printing
- Documenting VIs with the Show VI Info... Option

Customizing Your LabVIEW Environment
- Setting LabVIEW Preferences
- Customizing the Controls and Functions Palettes

Examples

LabVIEW comes with a large number of example VIs that are useful when learning to use LabVIEW. Use the VI called `readme.vi`, at the top level of the Examples directory, to look at the available examples. When you select a VI, the online documentation for that VI is displayed (this information was entered for the VI using the **Get Info...** dialog box). To open a VI, use the **Open** command from the **File** menu.

If you are going to use data acquisition, see the LabVIEW Data Acquisition Basics Manual, which contains very important information about using the DAQ VIs.

In addition, LabVIEW now includes the DAQ Navigator VI, which you access from the **Help** menu. This VI opens a series of dialog boxes that prompt you about the type of DAQ application you want to create. The VI then highlights and opens useful examples that illustrate how you might create your application, and also displays references in LabVIEW manuals that you can use for further research.

The DAQ examples directory contains a VI library called RUN_ME that has a getting started example VI for analog input, analog output, digital I/O, and counter/timers.

Analysis Examples

Support and Training

National Instruments offers several hands-on LabVIEW courses designed to save you and your company time and money by getting you up the learning curve and shortening your application development time. We offer the courses at our corporate headquarters in Austin, Texas, our international branch offices, regionally in cities around the world, or in your facility. Contact us for a course schedule, a detailed course outline, or to arrange a course at your facility.

Basics Course
Advanced Course
Data Acquisition Course
GPIB Course
VXI Course
LabVIEW Basics-Interactive!
LabVIEW Instructional Video

Note: We offer LabVIEW courses on PCs running Windows or on Macintoshs. Because LabVIEW is functionally similar on all platforms, the training offered pertains to other platforms.

Contact NI

Related Products

Instrument Drivers

An instrument driver is software that controls an instrument. LabVIEW is ideally suited to for creating instrument drivers because you can model the panel of the instrument on a VI front panel and use the block diagram to send the necessary control commands to the instrument. Therefore you no longer need to remember or type in the commands to control that instrument. You simply use the controls on the VI front panel. Controlling the instrument from a software front panel is of little value. The real advantage is that you can use the instrument driver as a subVI in conjunction with other subVIs in a larger VI to control an entire system. Use the link below to find out how to get LabVIEW instrument drivers.

To access the latest instrument drivers, you can use Bulletin Board Support, FTP Support, WWW Support, andFaxBack Order Form Support.

Add-On Toolkits

Each add-on toolkit adds application-specific functionality to the base LabVIEW system. Toolkits include VI libraries and complete documentation. Most toolkit VIs are delivered in LabVIEW block diagram source code, so you can make modifications to VIs as required by your application. The family of LabVIEW toolkits includes

Joint Time-Frequency Analysis
Picture Control
PID Control
Test Executive
SPC

Contact NI

Front Panel Reference

Front Panel Object Introduction
Importing graphics from Other Programs
Numeric Controls and Indicators
Boolean Controls and Indicators
String Controls and Indicators
Path Controls and Refnums
List and Ring Controls and Indicators
Arrays Controls and Indicators
Cluster Controls and Indicators
Graph and Chart Controls and Indicators

Block Diagram Reference

Block Diagram Introduction
Wiring the Block Diagram
Structures
Formula Node
Attribute Nodes
Global and Local Variables

Advanced Topics

Custom Controls and Type Definitions
Calling Code from Other Languages
Managing Applications
Understanding How LabVIEW Executes VIs
Portability Issues
Data Storage Formats
Performance Issues
- Performance Profiling
- Speeding Up Your VIs
- Memory Usage

Basics Course

This 3-day hands-on course, designed for new or novice users, shortens the initial learning curve and gives you a head start on your application. The course progresses from LabVIEW fundamentals and the construction of simple virtual instruments (VIs) to building a complete application involving data acquisition, analysis, and presentation.

Advanced Course

This 2-day hands-on course builds on topics covered in the Basics course and extends your understanding of LabVIEW so that you can take full advantage of its features, such as attribute nodes, local variables, the Control Editor, and bundle and unbundle by name functions. It also covers techniques to reduce memory requirements, optimize execution speed, and build the highest performance, most efficient VIs.

Data Acquisiton Course

This 2-day hands-on course is for users who want to use LabVIEW to control plug-in data acquisition (DAQ) boards. In the LabVIEW DAQ course, we teach you how to select the right components to implement a computer-based data acquisition and control system. Beginning with an introduction to transducers and signal conditioning, the course continues with how to correctly wire signals to your signal conditioning hardware or DAQ board to minimize noise and avoid ground loops. The course discusses all components of a DAQ system, including analog, digital, and counter/timer I/O. Practical hands-on exercises use thermocouples, strain gauges, and SCXI signal conditioning hardware connected to a multifunction DAQ board. We recommend that you have prior LabVIEW experience or attend the LabVIEW Basics course prior to attending this course.

Function and VI Reference

Functions are elementary nodes in the G programming language. They are analogous to operators or library functions in conventional languages. Functions are not VIs and therefore do not have front panels or block diagrams. When compiled, functions generate inline machine code.

You select functions from the **Functions** palette, shown in the following illustration. Click on an option for an explanation.

When the block diagram window is active, you can display the **Functions** palette by selecting **Windows»Show Controls and Functions**. You can also access the **Functions** palette by popping up on the area in the block diagram window where you want to place the function

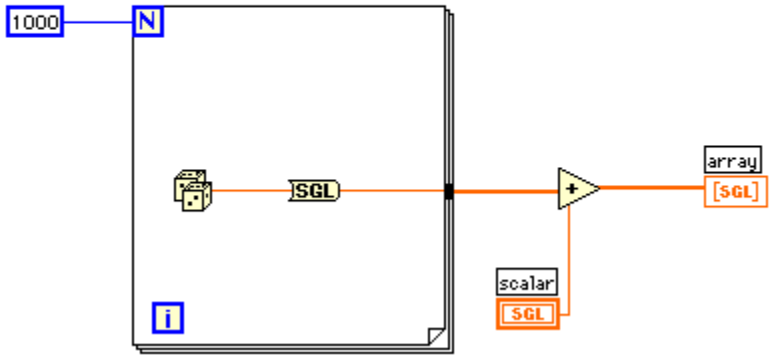
Introduction To Functions
Numeric Functions
Boolean Functions
String Functions
Array Functions
Cluster Functions
Comparison Functions
Time and Dialog Functions
File I/O Functions
Advanced Functions
Analysis VIs
Communications VIs
Data Acquisition VIs
Instrument I/O

Hints for Using this Help File

This file has different features depending on the platform you use it on. The common information is included here. Click on the platform you use for information specific to it.

This is a hypertext help file. That means you navigate by clicking on buttons, such as those at the top of this window and on clickable items, or hotspots, in the text. Topic hotspots are green and have a solid underline. Click here for an example. Pop-up hotspots are green and have a dotted underline. Click here for an example.

Many menu and palette pictures are clickable as well. If the cursor changes to a pointing hand over an element of a picture, you are over a hotspot. Click to see more information. Move the cursor over the following picture until the pointing hand appears and click



Some text is in red to set information off. Red does not denote a hotspot.

Windows 95 and UNIX help
Windows 3.x help
Macintosh and Power Macintosh help

Windows 95 and UNIX help

Windows 95 (and Windows NT 3.5.x) help files use a new interface featuring a contents, index and find tab. The contents tab gives you an expandable, collapsible hierarchical guide to all the topics in the help file. The index tab contains a keyword search engine that can span multiple help files in one help suite. The find tab is a full text search engine that you can use to find things not indexed. To learn how to use the features see the Windows 95 help file for a complete introduction.

The UNIX help compiler supports these new help features, so the above applies equally well to LabVIEW help on UNIX.

Example Topic

Clicking on these spots send you to a new topic, like this one. Click on the Back button above to go back to the hints topic

Pop-up Example

Information shows up in a floating window. Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

Example Graphic Hotspot Topic

Click anywhere to go back to the original topic.

VI Library Overview

Click on the group of VIs for which you want more overview information. Once there click on the overview link.

Analysis VIs
Communications VIs
Data Acquisition VIs
Instrument I/O

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.

Windows 3.x help

LabVIEW help has some added features beyond standard WinHelp. The Search button opens a keyword list that spans all the help files in the help file suite. The HyperView option in the File menu opens a expandable, collapsible hyerarchical guide to the topics in the help file. The Find+ button on the button bar (also available in the File menu) gives you a full text search engine complete with a very helpful topic preview.

Macintosh and Power Macintosh help

The Macintosh viewer provides you full text search capability with the Find command in the QuickHelp menubar. However, neither the full text search nor the keyword Search button work across the various help files in the LabVIEW help suite. The text or keyword must be in the help file currently open.

GPIB Course

This two day, hands-on GPIB course teaches you how to build, configure, and control GPIB applications using LabVIEW. This course includes information on using LabVIEW's GPIB VIs, serial polling devices, instrument drivers, and using GPIB to communicate with multiple computers.

VXI Course

This two day, hands-on VXI course teaches you how to build, configure, and control a VXIbus system using LabVIEW. This course teaches you to create and run applications that perform A/D data acquisition, waveform generation, counter/timer operations, and digital multimeter operations.

LabVIEW Basics-Interactive!

LabVIEW Basics - Interactive! is a self-paced multimedia course on CD-ROM that teaches LabVIEW concepts through an interactive user interface. Experience high quality video and audio as a National Instruments applications engineer builds example VIs onscreen. Based on material from the highly acclaimed *LabVIEW Basics* course, *LabVIEW Basics - Interactive!* progresses from LabVIEW fundamentals and the construction of simple VIs to developing data acquisition and instrument control applications.

LabVIEW Basics - Interactive! ships on a single CD-ROM that will run on PC's under Windows 95, Windows 3.1 or greater, and Macintosh/Power Macintosh running System 7.1 or greater. LabVIEW version 4.0 or greater must be installed separately.

LabVIEW Instructional Video

Using LabVIEW -Introduction is a two-hour instructional video that will help you to begin building your LabVIEW applications more quickly. This video is based on the LabVIEW Basics course, and it provides you with LabVIEW fundamentals, including dataflow programming, building simple VIs, creating subVIs, and using array and graphs.

Structures

For additional information, see Structures.

Numeric

Numeric Functions perform arithmetic operations, conversions, trigonometric, logarithmic, and complex mathematical functions. This palette also contains additional constants, such as Pi.

Boolean

Functions that use Boolean and logical functions.

String

String Functions manipulate strings and convert numbers to and from strings.

Array

Array Functions assemble, dissassemble, and process arrays.

Cluster

Cluster Functions assemble, dissassemble, and process clusters.

Comparison

Comparison Functions compare numbers or strings (greater than, less than, and so on) and functions that are based on a comparison, such as finding the minimum and maximum of two values.

Time & Dialog

Time and Dialog Functions contains functions that manipulate time and display dialog boxes. This palette also includes Error Handling VIs .

File I/O

File I/O Functions communicate with disk files.

Communication

Selecting **Functions»Communication** accesses the TCP/IP, DDE, UDP, PPC, AppleEvents, HiQ, OLE, Named Pipe, and System Exec VIs. See individual descriptions in Communication VIs .

Instrument I/O

Selecting **Functions»Instrument I/O** accesses the GPIB 488.2 VIs, the GPIB Traditional VIs, the Serial I/O VIs, and the VISA VIs. See individual descriptions in Instrument I/O VIs.

DAQ

Selecting **Functions»DAQ** accesses the Easy I/O, Analog Input, Analog Output, Counter, Digital Input and Output, Calibration and Config, Advanced, and DAQ Utility VIs. See individual descriptions in DAQ VIs.

Advanced

Advanced Functions are functions that do not fit into any other category, such as the Code Interface Node. This palette also contains Help window functions, VI control VIs, Data Manipulation functions, and Occurrences functions.

Analysis

Selecting **Functions»Analysis** accesses the Measurement, Signal Generation, Digital Signal Processing, Filters, Windows, Probability and Statistics, Curve Fitting, Linear Algebra, Array Operation, and Additional Numerical Methods VIs. If you are a Windows user and do not have the full developement system, click here for more information. Click here for descriptions of the Analysis VIs.

Select A VI...

When you select **Functions»Select a VI...**, LabVIEW opens a file dialog box. From there, you can select any VI library and VI you want to open.

Tutorial

Selecting **Functions»Tutorial** accesses the Tutorial VIs. You call these VIs while working through the LabVIEW Tutorial Manual.

Using Adobe Acrobat

First, you must install the reader for your platform. Consult the readme that comes with your reader.

Once the reader is installed, double click on either manual (CIRMBOOK.PDF for the Code Interface Reference Manual, or VXIBOOK.PDF for the LabVIEW VXI VI Reference Manual) to view it.

To hypernavigate, click on a bookmark from the list to the left of the book page. The expandable, collapsible bookmarks are like a hypertext table of contents. Use them to jump to any section in the book you like.

A bookmark with a triangle in front of it has at least one level of sub-bookmarks under it. Click on the triangle to reveal the lower level of bookmarks. Click on the triangle of the expanded bookmark again to collapse to the higher level.

You can also print the book if you want a hard copy.

Introduction To Functions

This topic is an overview of LabVIEW functions, which perform elementary operations such as computation, file I/O, and string formatting. The following topics provide some general information about functions: Polymorphism, Unit Polymorphism, Numeric Conversion, Overflow and Underflow, and Wire Styles.

Polymorphism

Polymorphism is the ability of a function to adjust to input data of different types or representations. Most functions are polymorphic. VIs are not polymorphic. All functions that take numeric input can accept any numeric representation (except some functions that do not accept complex numbers).

Functions are polymorphic to varying degrees; none, some, or all of their inputs may be polymorphic. Some function inputs accept numbers or Boolean values. Some accept numbers or strings. Some accept not only scalar numbers but also arrays of numbers, clusters of numbers, arrays of clusters of numbers, and so on. Some accept only one-dimensional arrays although the array elements may be of any type. Some functions accept all types of data, including complex numbers.

{bmc Polyc.BMP} The icon shown to the left indicates that a parameter is polymorphic. See individual function descriptions for details.

Unit Polymorphism

If you want to create a VI that computes the root, mean square value of a waveform, you have to define the unit associated with the waveform. You would need a separate VI for voltage waveforms, current waveforms, temperature waveforms, and so on. LabVIEW has polymorphic unit capability so that one VI can perform the same calculation, regardless of the units received by the inputs, .

You create a polymorphic unit by entering $*x*, where *x* is a number (for example, $1). You can think of this as a placeholder for the actual unit. When LabVIEW calls the VI, the program substitutes the units you pass in for all occurrences of $*x* in that VI.

LabVIEW treats a polymorphic unit as a unique unit. You cannot convert a polymorphic unit to any other unit, and polymorphic units propagate throughout the diagram, just as other units do. When the unit connects to an indicator that also has the abbreviation $1, the units match and the VI can then compile.

You can use $1 in combinations just like any other unit. For example, if the input is multiplied by 3 seconds and then wired to an indicator, the indicator must be $1 s units. If the indicator has different units, the block diagram shows a bad wire. If you need to use more than one polymorphic unit, you can use the abbreviations $2, $3, and so on.

A call to a subVI containing polymorphic units computes output units based on the units received by its inputs. For example, suppose you create a VI that has two inputs with the polymorphic units $1 and $2 that creates an output in the form $1 $2 / s. If a call to the VI receives inputs with the unit m/s to the $1 input and kg to the $2 input, LabVIEW computes the output unit as kg m / s^2.

Suppose a different VI has two inputs of $1 and $1/s, and computes an output of $1^2. If a call to this VI receives inputs of m/s to the $1 input and m/s^2 to the $1/s input, LabVIEW computes the output unit as m^2 / s^2. If this VI receives inputs of m to the $1 input and kg to the $1/s input, however, LabVIEW declares one of the inputs as a unit conflict and computes (if possible) the output from the other input.

A polymorphic VI can have a polymorphic subVI because LabVIEW keeps the respective units distinct.

Numeric Conversion

This section describes the rules for numeric conversion. You can convert any numeric representation to any other numeric representation. When you wire two or more numeric inputs of different representations to a function, the function usually returns output in the larger or wider format. The functions coerce the smaller representations to the widest representation before execution. The following list contains the LabVIEW icons for the available numeric data representations in order from smallest to largest.

{bmc Int8c.BMP} Byte integer

{bmc uint8c.BMP} Unsigned byte integer

{bmc int16c.BMP} Word integer

{bmc uint16c.BMP} Unsigned word integer

{bmc Int32c.BMP} Long integer

{bmc Uint32c.BMP} Unsigned long integer

{bmc Sglc.BMP} Single-precision floating-point number

{bmc Dblc.BMP} Double-precision floating-point number

{bmc Extc.BMP} Extended-precision floating-point number

Complex single-precision floating-point number
Complex double-precision floating-point number
Complex extended-precision floating-point number

Note*:* Unsigned integer represent only non-negative numbers, and have a larger range of positive numbers than signed integers even though the number of bits is the same for both representations.

Some functions, such as Divide, Sine, and Cosine, always produce floating-point output. If you wire integers to their inputs, these functions convert the integers to double-precision, floating-point numbers before performing the calculation.

For floating-point, scalar quantities, it is usually best to use double-precision, floating-point numbers. Single-precision, floating-point numbers save little memory, little or no time, and overflow much more easily. You should only use extended-precision, floating-point numbers when necessary. The performance and precision of extended-precision arithmetic varies among the platforms.

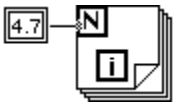
For integers, it is usually best to use a long integer.

If you wire an output to a destination that has a different numeric representation from the source, LabVIEW converts the data according to the following rules.

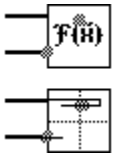
- Signed or unsigned integer to floating-point number--Conversion is exact, except for long integers to single-precision, floating-point numbers. In this case, LabVIEW reduces the precision from 32 bits to 24 bits.
- Floating-point number to signed or unsigned integer--LabVIEW moves out-of-range values to the integer's minimum or maximum value. In most integer objects, such as the iteration terminal of a For Loop, LabVIEW rounds floating-point numbers. LabVIEW rounds a fractional part of 0.5 to the nearest even integer--for example, LabVIEW rounds 6.5 to 6 rather than 7.
- Integer to integer--LabVIEW does not move out-of-range values to the integerÕs minimum or maximum value. If the source is smaller than the destination, LabVIEW extends the sign of a signed source and places zeros in the extra bits of an unsigned source. If the source is larger than the

destination, LabVIEW copies only the low order bits of the value.

On the block diagram, LabVIEW places a *coercion dot* on the border of a terminal where the conversion takes place to indicate that automatic numeric conversion occurred, as in the following example.



Because VIs and functions can have many terminals, a coercion dot can appear inside an icon if the wire crosses an internal terminal boundary before it leaves the icon/connector, as the following illustration shows. Moving a wired icon stretches the wire.



Coercion dots can cause a VI to use more memory and time. You should try to keep data types consistent in your VIs. For more information on coercion dots, see Chapter 8, Customizing Your LabVIEW Environment, in the LabVIEW User Manual.

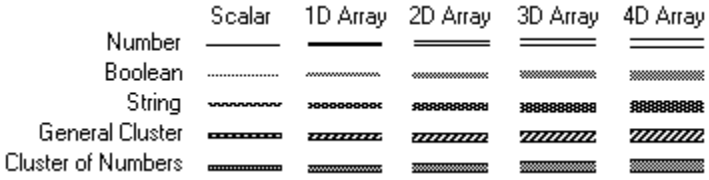
Overflow and Underflow

LabVIEW does not check for overflow or underflow conditions on integer values. Overflow and underflow for floating-point numbers is in accordance with IEEE 488 Standard 754 for binary, floating-point arithmetic.

Floating-point operations propagate NaN and ±Inf. When you explicitly or implicitly convert NaN or ±Inf to an integer or Boolean value, however, you get a value that looks reasonable, but is meaningless. For example, dividing by zero produces ±Inf, but converting that value to a word integer gives the value 32,767 or -32,768, which is the largest value that can be represented in the destination format.

Wire Styles

The wire style represents the data type for each terminal, as the following table shows. Polymorphic functions show the wire style for the most commonly used data type.



Structures

For additional information, see Structures.

Numeric

Numeric Functions perform arithmetic operations, conversions, trigonometric, logarithmic, and complex mathematical functions. This palette also contains additional constants, such as Pi.

Boolean

Boolean Functions consists of logical functions.

String

String Functions manipulate strings and convert numbers to and from strings.

Array

Array Functions assemble, dissassemble, and process arrays.

Cluster

Cluster Functions assemble, dissassemble, and process clusters.

Comparison

Comparison Functions compare numbers or strings (greater than, less than, and so on) and functions that are based on a comparison, such as finding the minimum and maximum of two values.

Time & Dialog

Time and Dialog Functions manipulate time, display dialog boxes, and handle errors handle errors .

File I/O

File I/O Functions communicate with disk files.

Communication

Access to the Communications VIs--TCP/IP, DDE, UDP, PPC, HiQ, OLE, Named Pipes, and System Exec VIs. For more information, see Communication VIs.

Instrument I/O

Access to the GPIB 488.2 VIs, the GPIB Traditional VIs, the Serial I/O VIs, and the VISA VIs. For more information, see Instrument I/O VIs.

DAQ

Access to the DAQ VI library palette. This library includes the following VI subpalettes--Easy I/O, Analog Input, Analog Output, Counter, Digital Input and Output, Calibration and Config, Advanced, and DAQ Utilities. For more information, see DAQ VIs.

Advanced

Advanced Functions are functions that do not fit into any other category, such as the Code Interface Node. This palette also contains Help window functions, VI Control VIs, Data Manipulation functions, and Occurrences functions.

Analysis

Access to the Analysis VI library palette. The full developement system includes the following VI subpalettes-- Signal Generation, Digital Signal Processing, Measurement, Filters, Windows, Probability & Statistics, Curve Fitting, Linear Algebra, Array Operations, and Additional Numerical Methods VIs. If you are a Windows user and do not have the full developement system, click here for more information. For more information, see Analysis VIs.

Select A VI...

When you select **Functions»Select a VI...**, LabVIEW opens a file dialog box. From there, you can select any VI library and VI you want to open.

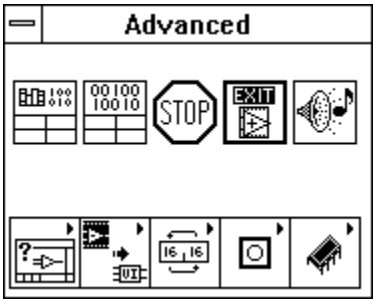
Tutorial

Selecting **Functions»Tutorial** accesses the Tutorial VIs. You call these VIs while working through your LabVIEW Tutorial Manual.

Advanced Functions

This topic describes the functions that perform advanced operations. This topic also describes the Help window Functions, VI controls, data manipulation, occurrences, and memory functions. For general information about Advanced functions, see Polymorphism for Advanced Functions .

To access the **Advanced** palette, shown in the following illustration, select **Functions»Advanced**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Beep
Code Interface Node
Call Library Function
Quit LabVIEW
Stop

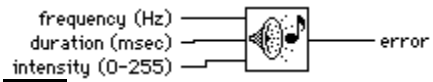
Subpalettes

Data Manipulation
Help
Memory
Occurrences
VI Control

For examples of CIN functions, see `examples\cin`.

Beep

Causes the system to issue an audible tone. You can specify the tone frequency in Hertz, the duration in milliseconds, and the intensity as a value from 0 to 255, with 255 being the loudest. Although this VI appears on all LabVIEW platforms, the frequency, duration, and intensity parameters work only on the Macintosh.

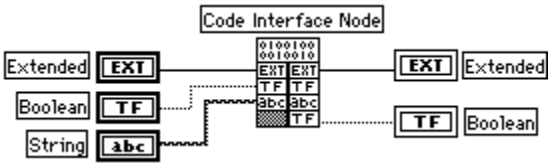


DBL **frequency (Hz)** is the frequency of the tone in Hertz.
I16 **duration (msec)** is the duration of the intensity of the tone.
I16 **error** returns an error.

Code Interface Node

With a CIN, you can call code written in a conventional programming language, such as C, directly from a block diagram. CINs make it possible for you to use algorithms written in another language or to access platform-specific features or hardware that LabVIEW does not directly support.

Code Interface Nodes are resizable and show datatypes for the connected inputs and outputs, similar to the Bundle function. The following illustration shows the CIN function.



LabVIEW's interface to external code is very powerful. You can pass any number of parameters to or from external code, and each parameter can be of arbitrary, LabVIEW datatype. LabVIEW provides several libraries of routines that make working with LabVIEW datatypes easier. These routines support memory allocation, file manipulation, and datatype conversion.

If you convert a VI that contains a CIN to another platform, you need to recompile the code for the new platform, because CINs use code compiled in another programming language. You can write source code for a CIN so that it is machine-independent, requiring only a recompile to convert it to another computer.

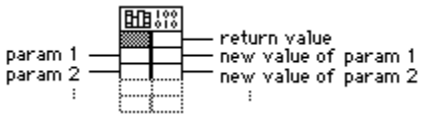
For help with CIN related functions see the Code Interface Function Reference

Call Library Function

With the Call Library Function node, you can call standard libraries without writing a Code Interface Node (CIN). Under Windows, you can call a dynamic link library (DLL) function directly. In Macintosh and Unix, you can call a shared library function directly. On the Macintosh 68K, you must have the CFM-68K system extension installed for the Call Library Function node to operate.

This node supports a large number of datatypes and calling conventions. You should be able to use it to call functions from most standard and custom-made libraries.

The Call Library Function node, shown in the following illustration, looks similar to a Code Interface Node.



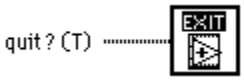
The Call Library Function consists of paired input/output terminals with input on the left and output on the right. You can use one or both. The return value for the function is returned in the right terminal of the top pair of terminals of the node. If there is no return value, then this pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the functions parameter list. You pass a value to the function by wiring to the left terminal of a terminal pair. You read the value of a parameter after the function call by wiring from the right terminal of a terminal pair.

If you select Configure... from the pop-up menu of the node, LabVIEW displays a Call Library Function dialog box from which you can specify the library name or path, function name, calling conventions, parameters, and return value for the node. When you click the OK, the node automatically increases in size to have the correct number of terminals. It then sets the terminals to the correct datatypes.

For more information on this function, see Calling Code from Other Languages.

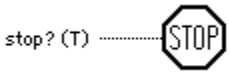
Quit LabVIEW

Stops all executing VIs and ends the current session of LabVIEW. This function shuts down only LabVIEW; the function does not power down the system or affect other applications. The function stops running VIs the same way the Stop function does.



Stop

Stops the VI in which it executes, just as if you clicked the stop button in the toolbar.



stop? (T). If you wired the input, stop occurs only if the input value is TRUE. If you leave the input unwired, the stop occurs as soon as the node that is currently executing finishes.

If you need to abort execution of all VIs in a hierarchy from the block diagram, you can use this function, but *you must use it with caution.* Before you call the Stop function with a TRUE input, be sure to complete all final tasks for the VI first, such as closing files, setting save values for devices being controlled, and so on. If you put the Stop function in a subVI, you should make its behavior clear to other users of the VI, because this function causes their VI hierarchies to abort execution.

In general, you should avoid using the Stop function when you have a built-in terminator protocol in your VI. For example, I/O operations should be performed in While Loops so that the VI can terminate the loop on an I/O error. You should also consider using a front panel Stop Boolean control to terminate the loop at the request of the user rather than using the Stop function.

Polymorphism for Advanced Functions

Some of the advanced functions are polymorphic--they work on inputs of more than one representation or data structure.

For functions that work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on, a formal and recursive definition of the allowable input type is as follows.

> *Numeric type* = numeric scalar || array [*numeric type*] || cluster [*numeric types*]
>
> where numeric scalars can be floating-point, integers, or complex, but arrays of arrays are not allowed.

Arrays can have any number of dimensions and any size. Clusters can have any number of elements. The output type is of the same composition as the input type, and the functions operate on each element of the structure, for functions with one input.

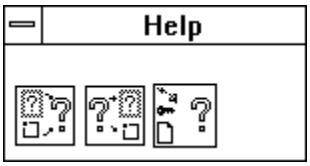
For functions with two inputs, LabVIEW allows the following input combinations.

- *Similar*--both inputs have the same structure, and the output has the same structure as the inputs.
- *One scalar*--one input is a numeric scalar, the other is an array or cluster, and the output is an array or cluster.
- *Array of*--one input is a numeric array, the other is that numeric type, and the output is an array.

For similar inputs, LabVIEW performs the function on the respective elements of the structures. Both arrays must have the same dimension and for predictable results must have the same dimension size and number of elements. Both clusters must have the same number of elements, and the respective elements must have the same structure. For operations involving a scalar and an array or cluster, LabVIEW performs the function on the scalar and the respective elements of the structure. For operations that involve a numeric type and an array of that type, LabVIEW performs the function on each array element.

Help Function Descriptions

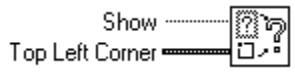
The following illustration displays the options available on the **Help** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Control Help Window
Control Online Help
Get Help Window Status

Control Help Window

Modifies the Help window by showing, hiding, or by repositioning the window.

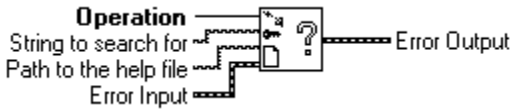


Show. Pass in TRUE to show the window, FALSE to hide the window.

Top Left Corner consists of a cluster of numbers, which represent a point. These are offset from the upper left corner of the screen where the Help window should appear.

Control Online Help

Modifies the online help file, which you can use to display the table of contents, jump to a specific point in the file, or close the online help.



Operation is an enumeration that is one of contents, key, and close.

String to search for specifies the tag in the help file to display.

Path to the help file.

error input describes error conditions occurring before this function executes. If an error has already occurred, this function returns the value of the **error input** cluster in **error output**. The function executes normally only if no incoming error exists; otherwise it merely passes the **error input** value to **error output**.

status is TRUE if an error occurred. If **status** is TRUE, this function does not perform any operations.

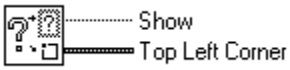
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the function that produced the error.

error output contains **error in**formation. If the **error input** cluster indicated an error, the **error output** cluster contains the same information. Otherwise, **error output** describes the error status of this function.

Get Help Window Status

Returns the status and the position information for the Help window.



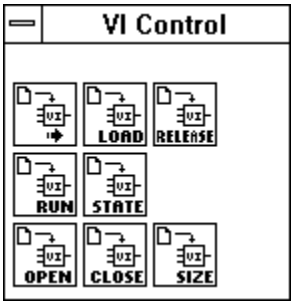
Show. Set to TRUE if the Help window is displayed, FALSE if the Help window is hidden or iconified.

Top Left Corner consists of a cluster of numbers, which represent a point. These are offset from the upper left corner of the screen where the Help window should appear.

VI Control VI Descriptions

You can use the VI Control VIs to dynamically load, call, and close other VIs. When you call a VI dynamically, you specify whether or not the called VI opens its front panel and then closes the front panel when it finishes executing. You can also pass parameters to and from the dynamically called VI.

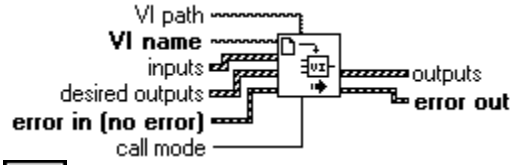
All of these VIs use error cluster inputs and outputs to make error handling easier. If an incoming error is set, the VI does not do anything. The Release Instrument VI, however, releases the specified VI regardless of incoming errors. The following illustration displays the options available on the **VI Control** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Call Instrument
Close Panel
Open Panel
Preload Instrument
Release Instrument
Resize Panel
Run Instrument

Call Instrument

Loads and then calls another VI as long as the VI you are calling is not currently in use by your main VI. For example, if you have the Serial Port Read VI on your block diagram, you cannot use Call Instrument to call Serial Port Read directly, because it is already in the main VIs hierarchy. However, you can call the Serial Port Read VI if you create a VI that is not part of the main VIs hierarchy. If the called VI has not already been loaded, LabVIEW loads it before the call, and unloads the VI when the call is finished. If you do not want to incur the speed penalty of loading the VI at the time of the call, use the Preload Instrument VI to preload the VI, and then use the Release Instrument VI when you are finished with it. If **error in** contains an error, LabVIEW does not call the VI.



VI Name is used for all operations. Case sensitivity of the name match is the same as that of the file system convention of the platform you are using.

VI path specifies the path of the directory or library containing the VI you want to call (not including the VI name). If the VI is already in memory, you do not need to specify the path.

inputs is an array of clusters containing a control name, a type descriptor, and flattened data. You can get the type descriptor and the flattened data from the Flatten to String function.

Note: You can pass data to any control (excluding indicators) on the front panel of the called VI; the controls do not have to be on the connector pane of the called VI.

desired outputs contain the same datatype as **inputs**. If you do not wire **desired outputs**, all indicators on the front panel of the VI are returned in **outputs**, using the front panel order. If you wire **desired outputs**, the name component of each element must specify the name of a valid indicator. LabVIEW ignores the type descriptor and data components of the clusters.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

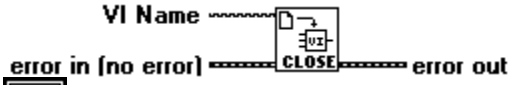
call mode is used to specify whether the called VI opens its front panel when called and then closes its front panel if the front panel was originally closed. If you specify a value of 0, the front panel window does not open. If you specify a value of 1, LabVIEW opens the front panel and then closes it if the front panel was originally closed.

outputs contains the same datatype as **inputs**. If you do not wire **outputs**, all indicators on the front panel of the called VI are returned in **outputs**, in front panel order. If you do wire **outputs**, the name component of each element must specify the name of a valid indicator. LabVIEW ignores the type descriptor and data components of the clusters.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Close Panel

Closes the frontpanel of a specified VI.



VI Name is the name of the VI whose front panel you want to close. Case sensitivity of the name match is the same as that of the file system convention on the platform you are using.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

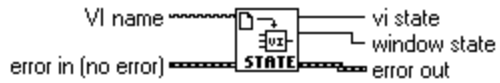
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Get Instrument State

You can use this VI to return the execution state of a VI and the state of the front panel. If the VI is not in memory, then a File Not Found error will be returned.



VI name is the name of the VI from which you want the execution state and the front panel state. Case sensitivity of the name match is the same as that of the platforms file system.

error in (no error) describes error conditions occurring before this VI executes. If an error has

already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

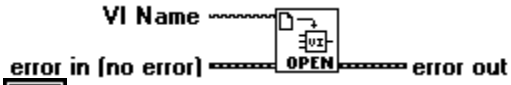
VI state is the execution state of the VI. The three possible values are Broken, Idle, or Running. Broken means that the VI has errors that make it nonexecutable. Idle means that the VI is not running, but the VI is in memory. Running means that the VI is presently executing.

Window state is the state of the front panel window. The three possible values are Closed, Open, or Open and Active (window).

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Open Panel

Opens the front panel of the specified VI. The specified VI must already be in memory, either because it was loaded using the Preload Instrument VI, or because it is the subVI of another VI.



VI Name is the name of the VI whose front panel you want to open. Case sensitivity of the name match is the same as that of the file system convention of the platform you are using.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

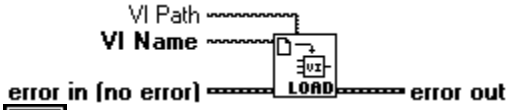
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Preload Instrument

You can use this VI to load another VI into memory. The front panel of the specified VI is not visible when it is loaded. If you want the front panel to be visible, call either Load Panel VI or use the appropriate call mode for the Call Instrument VI.



VI Name is the name of the VI you want to load. Case sensitivity of the name match is the same as that of the file system convention of the platform you are using.

VI path is the absolute path to the VI to be loaded, not including **VI Name**.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

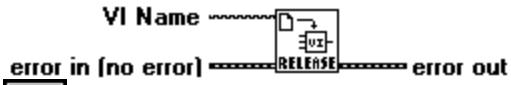
source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

If you execute a Preload Instrument VI, and it does not return an error, make sure you call the Release Instrument VI when you are finished to remove the loaded VI from memory. If you call the Preload Instrument VI multiple times, you need to balance the calls that the Release Instrument VI calls.

Release Instrument

Use this VI to unload a VI that was loaded using the Preload Instrument VI. If you call Preload Instrument more than once; the specified VI is not unloaded until you call Release Instrument an equal number of times.



VI Name is the name of the VI you want to release. Case sensitivity of the name match is the same as that of the file system convention on the platform you are using.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

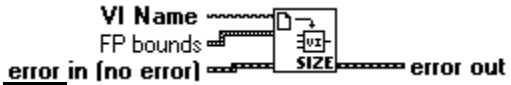
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Resize Panel

Resizes and/or moves the front panel of a VI that is already in memory. The VI must be in memory, but its front panel does not have to be open. Consequently, you can size or position a front panel before opening it.



VI Name is the name of the VI whose front panel you want to close. Case sensitivity of the name match is the same as that of the file system convention on the platform you are using.

FP bounds is the cluster that describes the top, left, bottom, and right edges of a bounding rectangle for the front panel, not including the window title bar. The minimum width and height for any of these parameters is 90.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

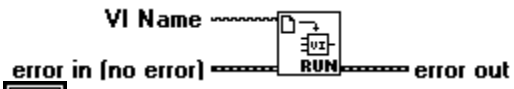
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Run Instrument

You can use this VI to run another VI that is in memory with the front panel of the VI in memory open. Run Instrument is different from Call Instrument in that Run Instrument returns immediately after starting the specified VI running, whereas Call Instrument waits for the called VI to complete execution and can pass parameters to and from the called VI. Run Instrument works just as if you selected **Operate»Run**, whereas Call Instrument functions more like a subVI call.



VI Name is used for all operations. Case sensitivity of the name match is the same as that of file system convention of the platform you are using.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

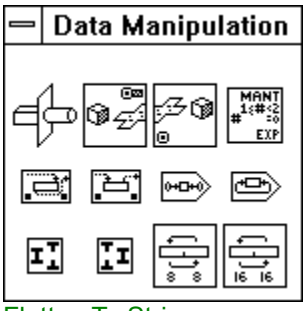
code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI that produced the error.

error out contains **error in**formation. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Data Manipulation Function Descriptions

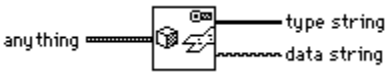
The following illustration displays the options available on the **Data Manipulation** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Flatten To String
Join Numbers
Logical Shift
Mantissa & Exponent
Rotate
Rotate Left With Carry
Rotate Right With Carry
Split Number
Swap Bytes
Swap Words
Type Cast
Unflatten From String

Flatten To String

Converts **anything** to a string of binary values.



POLY **anything** can be any data\type.

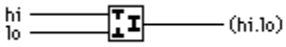
I16 **type string** contains the binary type descriptor for the data string. **type string** is an encoded representation of the datatype. In addition, the embedded 32-bit quantities used as array header information appear in the array with the most significant word first. See Data Storage Formats for more information.

abc **data string**. Usually, LabVIEW stores data as noncontiguous, indirectly referenced pieces. This function copies the data in LabVIEW form into a contiguous buffer **data string**. In addition, **data string** contains header information before each nonscalar component describing its size. Such a string can be stored in a file or sent over a serial line. If you send the string over a serial line, the receiver must be able to interpret it.

The elements of **data string** are in machine-independent, big endian form. That is, the function puts the most significant byte or word first and the least significant byte or word last, removes alignment, and converts extended-precision numbers to 16 bytes.

Join Numbers

Creates a number from the component bytes or words.

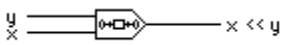


hi and **lo** can be 8-bit or 16-bit numbers or an array or cluster of those representations.

(hi.lo) is of the same data structure as **hi**. **(hi.lo)** is a 16-bit or 32-bit unsigned integer, or an array or cluster of those representations.

Logical Shift

Shifts **x** the number of bits specified by **y**.



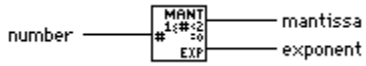
y can be any numeric representation. If **y** is greater than 0, the function shifts **x** left **y** bits (from least significant to most significant bit) and inserts zeros in the low-order bits. If **y** is less than 0, the function shifts **x** right **y** bits (from most significant to least significant bit) and inserts zeros in the high-order bits. If **y** is a floating point number, the function rounds it to an integer value.

x can be any numeric representation. If **x** is a byte, word, or long integer and **y** is greater than 8, 16, or 32 or is less than -8, -16, or -32, respectively, then the output value is all zeros. If **x** is a floating-point number, the function converts it to a long integer before shifting. See the Polymorphism for Advanced Functions topic for more information.

x << y.

Mantissa & Exponent

Returns the mantissa and exponent of the input numeric value such that **number** = **mantissa** * 2**exponent**. If **number** is 0, both **mantissa** and **exponent** are 0. Otherwise, the value of **mantissa** is greater than or equal to 1 and less than 2, and the value of **exponent** is an integer.



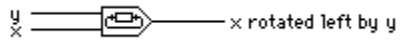
number can be any numeric representation. See the Polymorphism for Advanced Functions topic for more information.

mantissa has the same numeric representation as **number**.

exponent has the same numeric representation as **number**.

Rotate

Rotates **x** the number of bits specified by **y**.



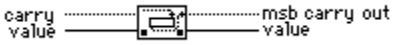
y. If **y** is greater than 0, the function rotates **x** left **y** bits (from least significant to most significant bit) and inserts the high-order bits in the low-order bits. If **y** is less than 0, the function rotates **x** right **y** bits (from most significant to least significant bit) and inserts the low-order bits in the high-order bits. If **y** is a floating-point number, it is rounded to an integer value.

x. If **x** is a floating-point number, the function rounds it to a long integer before rotating. If **x** is a byte, word, or long integer, then for any value of **y**, **y** ± 8, **y** ± 16, or **y** ± 32 yields the same output value, respectively, as **y**. For example, if **x** is a byte integer, then **y** = 1 and **y** = 9 yield the same result. See the Polymorphism for Advanced Functions topic for more information.

x rotated left by y.

Rotate Left With Carry

Rotates each bit in the input **value** to the left (from least significant to most significant bit), inserts **carry** in the low-order bit, and returns the most significant bit.



value must be a numeric scalar. It cannot be an array or a cluster. If **value** is a floating point number, the function rounds it to a long integer before rotating. See the Polymorphism for Advanced Functions topic for more information.

carry.

value is the new value.

msb carry out is the high-order bit of **value**.

Rotate Right With Carry

Rotates each bit in **value** to the right (from most significant to least significant), inserts **carry** in the high-order bit, and returns the least significant bit.

value must be a numeric scalar. It cannot be an array or a cluster. If **value** is a floating point number, the function rounds it to a long integer before rotating. See the Polymorphism for Advanced Functions topic for more information.

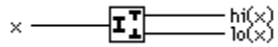
carry.

value is the new value.

lsb carry out is the low-order bit of **value**.

Split Number

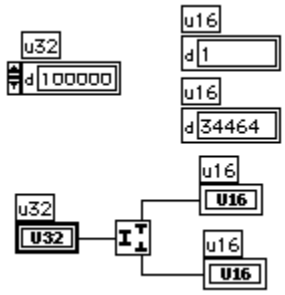
Breaks a number into its component bytes or words.



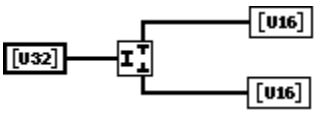
x can be a 16- or 32-bit integer, or an array or cluster of those representations.

hi(x) and **lo(x)** are of the same data structure as **x**. **hi(x)** and **lo(x)** are 8-bit and 16-bit unsigned integers, respectively, or an array or cluster of those representations.

The following illustration shows an example of how to use the Split Number function. The function splits the signed 32-bit number 100,000 into the high word component, 1, and the low word component, 34,464.



The following illustration shows the Split Number function operating on an array of unsigned, 32-bit integers.



Swap Bytes

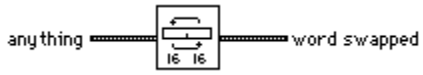
Swaps the high-order 8 bits and the low-order 8 bits for every word in **anything**.

anything. This function swaps the high and low bytes of any 16-bit integer or unpacked Boolean and the bytes of *each word* in a 32-bit integer. Strings, floating point numbers, and 8-bit integers are unaffected. See the Polymorphism for Advanced Functions topic for more information.

byte swapped is of the same datatype as **anything**.

Swap Words

Swaps the high-order 16 bits and the low-order 16 bits for every long integer in **anything**.

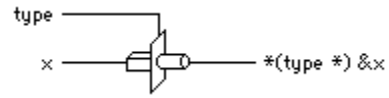


anything. This function swaps the high and low words of a 32-bit integer. Strings, floating-point numbers, 8-bit integers, and 16-bit integers are unaffected. See the Polymorphism for Advanced Functions topic for more information.

word swapped is of the same datatype as **anything**.

Type Cast

Casts **x** to the datatype, **type**.



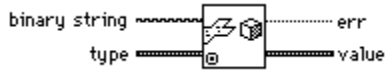
x and **type**. You can use these functions to convert a 1D array of flat data to a string or a string to a 1D array of flat data. Flat data is defined recursively as a numeric scalar, a Boolean scalar, or a cluster of flat data. See the Polymorphism for Advanced Functions topic for more information.

***(type *)&x** is of the same datatype as **type** and **x**.

Casting data to a string converts it into machine-independent, big endian form. That is, the function puts the most significant byte or word first and the least significant byte or word last, removes alignment, and converts extended-precision numbers to 16 bytes. Casting a string to a 1D array converts the string from machine-independent form to the native form for that platform.

Unflatten From String

Converts **binary string** to the type wired to **type**. This function performs the inverse of Flatten To String.



binary string.

type is a normally constructed LabVIEW type, not the type descriptor string output in the Flatten To String function. See Data Storage Formats.

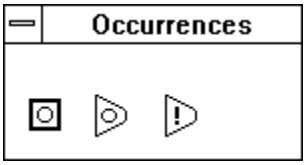
err is TRUE if the conversion is unsuccessful.

value is of the same datatype wired to **type**.

Occurrences Function Descriptions

You can use the occurrence functions to control separate, synchronous activities. In particular, you use these functions when you want one VI or part of a block diagram to wait until another VI or part of a block diagram finishes a task without forcing LabVIEW to poll.

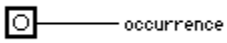
You can perform the same task using global variables, with one loop polling the value of the global until its value changes. However, global variables use more overhead, because the loop that waits uses execution time. With occurrences, the second loop can becomes idle and does not use processor time. When the first loop sets the occurrence, LabVIEW activates the second loop and any other block diagrams that wait for the specified occurrence. The following illustration displays the options available on the **Occurrences** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Generate Occurrence
Set Occurrence
Wait On Occurrence

Generate Occurrence

Creates an **occurrence** that you can pass to the Wait on Occurrence and Set Occurrence functions.

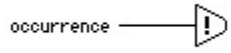


 **occurrence** is the occurrence refnum that links Wait on Occurrence and Set Occurrence. Ordinarily, only one Generate Occurrence node is connected to any set of Wait on Occurrences and Set Occurrences. You can connect a Generate Occurrence node to any number of Wait on Occurrence and Set Occurrence nodes. You do not have to have the same number of Wait on Occurrence and Set Occurrence nodes.

Each Generate Occurrence node on a block diagram represents a single, unique occurrence. In this way, you can think of the Generate Occurrence function as a constant. When a VI is running, every time a Generate Occurrence node executes, the node produces the same value. For example, if you place a Generate Occurrence node inside of a loop, the value produced by Generate Occurrence is the same for every iteration of the loop. If you place a Generate Occurrence node on the block diagram of a reentrant VI, Generate Occurrence produces a different value for each caller.

Set Occurrence

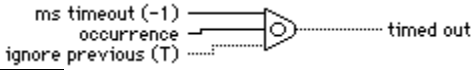
Triggers the specified **occurrence**. All block diagrams that are waiting for this occurrence stop waiting.



 **occurrence**.

Wait On Occurrence

Waits for the Set Occurrence function to set or trigger the given **occurrence**.



ms timeout (-1). If the occurrence does not occur within the specified **ms timeout**, measured in milliseconds, the function returns a value of TRUE. If **ms timeout** is -1, the function does not time out.

occurrence.

ignore previous (T). If **ignore previous** is TRUE, and another node has set the occurrence before this function began executing, then the function ignores the previous occurrence and waits for another one.

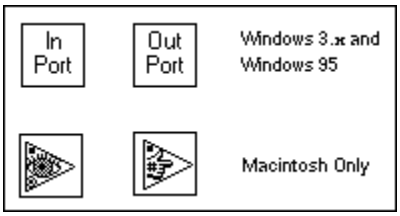
If **ignore previous** is FALSE, this function sets immediately if the occurrence was set since the last time this function waited on the incoming occurrence. This function remembers only the previous occurrence it waited on. If **ignore previous** is FALSE, and one or more occurrences are set between calls to this function, the function stops program execution and immediately returns with a timed out FALSE. Otherwise, it waits until you set an occurrence or exceed the time limit.

For example, suppose that you set three occurrences between successive calls to this function. The function stops program execution and immediately returns a timed out FALSE indicating that on the second call to signal that one or more occurrences have been set. If no occurrences are pending when LabVIEW calls the function, it waits until you set an occurrence or a timeout occurs.

timed out is TRUE if the occurrence does not occur within the specified **ms timeout**. If **ms timeout** is -1, **timed out** is FALSE.

Memory VI Descriptions (Windows and Macintosh)

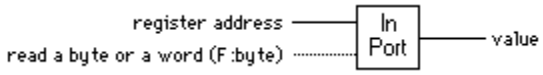
The following illustration displays the options available on the **Memory** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



In Port (Windows)
Out Port (Windows)
Peek (Macintosh)
Poke (Macintosh)

In Port (Windows)

Reads a byte or word integer from a specific register address. Because this VI is not available on all platforms, LabVIEW programs using this VI are not portable.



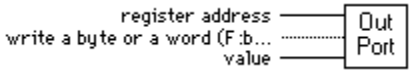
U32 **register address** specifies the address from which you want the byte or word integer to be read.

abc **read a byte or a word (F:byte)** specifies whether the information read from the register address is a byte (FALSE) or a word (TRUE). The default is FALSE and byte.

U16 **value** specifies the return value of the byte or word.

Out Port (Windows)

Writes a byte or word integer to a specific register address. Because this VI is not available on all platforms, LabVIEW programs using this VI are not portable.



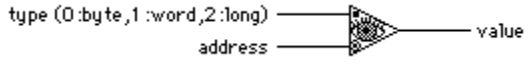
abc **register address** specifies the address to which you want the byte or word integer sent.

abc **write a byte or a word (F:b...)** specifies whether the information to be ported is a byte (FALSE) or a word (TRUE). The default is FALSE and byte.

U16 **value** specifies the input value of the byte or word.

Peek (Macintosh)

Reads a byte, word, or long integer value from a specific address. Because this VI is not available on other platforms, LabVIEW programs using this VI are not portable.



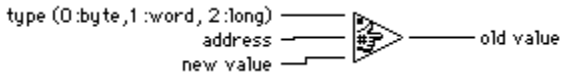
abc **type (0: byte, 1:word, 2:long)** specifies whether the integer value is a byte (0), a word (1), or a long integer (2).

abc **address** is the address from which you want the byte, word, or long integer value read.

U32 **value** is the return value of the byte, word, or long integer.

Poke (Macintosh)

Writes a byte, word, or long integer value to a specific address. Because this VI is not available on other platforms, LabVIEW programs using this VI are not portable.



type (0: byte, 1:word, 2:long) specifies whether the integer value is a byte (0), a word (1), or a long integer (2).

address is the address to which you want the byte, word, or long integer written.

new value is the new or input value of the byte, word, or long integer.

old value is the old or return value of the byte, word, or long integer.

Code Interface Node Function

Code Interface Node

Call Library Function (Windows, Macintosh, and Unix)

Call Library Function

Quit LabVIEW Function

Quit LabVIEW

Stop Function

Stop

Help Subpalette

Help Function Descriptions

VI Control Subpalette

VI Control VI Descriptions

Data Manipulation Subpalette

Data Manipulation Function Descriptions

Occurrences Subpalette

Occurrences Functions Descriptions

Control Help Window Function

Control Help Window

Control Online Help Function

Control Online Help

Get Help Window Status Function

Get Help Window Status

Call Instrument.vi

Call Instrument

Close Panel.vi

Close Panel

Get Instrument State.vi

Get Instrument State

Open Panel.vi

Open Panel

Preload Instrument.vi

Preload Instrument

Release Instrument.vi

Release Instrument

Resize Panel.vi

Resize Panel

Run Instrument.vi

Run Instrument

Flatten To String Function

Flatten To String

Join Numbers Function

Join Numbers

Logical Shift Function

Logical Shift

Mantissa & Exponent Function

Mantissa & Exponent

Rotate Function

Rotate

Rotate Left With Carry Function

Rotate Left With Carry

Rotate Right With Carry Function

Rotate Right With Carry

Split Number Function

Split Number

Swap Bytes Function

Swap Bytes

Swap Words Function

Swap Words

Type Cast Function

Type Cast

Unflatten From String Function

Unflatten From String

Generate Occurrence Function

Generate Occurrence

Set Occurrence Function

Set Occurrence

Wait On Occurrence Function

Wait On Occurrence

Beep Function

Beep

In Port (Windows) Function

In Port (Windows)

Out Port (Windows) Function

Out Port (Windows)

Peek (Macintosh) Function

Peek (Macintosh)

Poke (Macintosh) Function

Poke (Macintosh)

Memory Subpalette

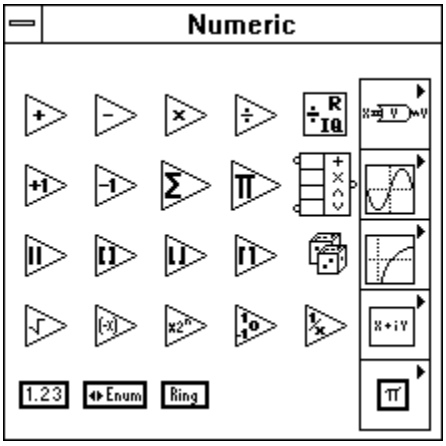
Memory VIs

Numeric Function Descriptions

Click here to access the Polymorphism for Numeric Functions topic.

This topic describes the functions that perform arithmetic operations, complex ,conversion, logarithmic , and trigonometric operations. This topic describes the commonly used constants like the numeric constant, enumerated constant, and ring constant as well additional numeric constants.

To access the **Numeric** palette, select **Functions»Numeric**. The following illustration shows the options that are available on the **Numeric** palette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Absolute Value
Add
Add Array Elements
Compound Arithmetic
Decrement
Divide
Enumerated Constant
Increment
Multiply
Multiply Array Elements
Negate
Numeric Constant
Quotient & Remainder
Random Number (0-1)
Reciprocal
Ring Constant
Round To +Infinity
Round To -Infinity
Round To Nearest
Scale By Power Of 2
Sign
Square Root
Subtract

Subpalettes

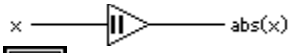
Additional Numeric Constants
Complex

Conversion
Logarithmic
Trigonometric

For examples of some of the arithmetic functions, see `examples\general\structs.llb`.

Absolute Value

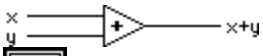
Returns the absolute value of the input.



x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

abs(x).

Add

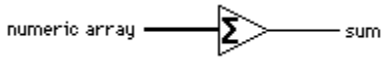


x and **y** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x+y.

Add Array Elements

Returns the sum of all the elements in **numeric array.**

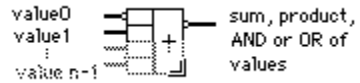


numeric array can have any number of dimensions. See Polymorphism for Numeric Functions for more information.

sum is of the same data type as the elements in **numeric array**.

Compound Arithmetic

Performs arithmetic on two or more numeric, cluster, or boolean inputs.



value 0...n can be a number or Boolean, array of numbers or Booleans, a cluster, array of clusters, and so on. See Polymorphism for Numeric Functions for more information.

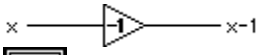
sum, product, AND, or OR of values returns the sum, product, AND, or OR value inputs. You select the operation (multiply, AND, or OR) by popping up on the function and selecting **Change Mode**.

You can invert the inputs or the output of this function by popping up on the individual terminals, and selecting **Invert**. For add, select **Invert** to negate an input or the output. For multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For AND or OR, select **Invert** to logically negate an input or the output.

Note: You add inputs to this node by popping up on an input and selecting Add Input or by placing the Positioning tool in the lower left or right corner of the node and dragging it.

Decrement

Subtracts 1 from the input value.



x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See Polymorphism for Numeric Functions for more information.

x-1.

Divide

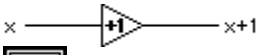
Computes the quotient of the inputs.

x and **y** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x/y is a double-precision floating-point number if both **x** and **y** are integers. In general, the output type is the widest representation of the inputs if the inputs are not integers or if their representations differ.

Increment

Adds 1 to the input value.

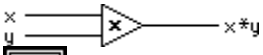


x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x+1.

Multiply

Returns the product of the inputs.

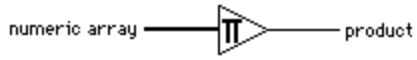


x and **y** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x*y.

Multiply Array Elements

Returns the product of all the elements in **numeric array.**

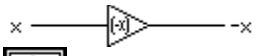


numeric array can have any number of dimensions. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

product is of the same data type as the elements in **numeric array**.

Negate

Negates the input value.

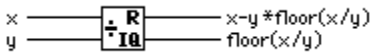


x can be a scalar signed number, array or cluster of signed numbers, array of clusters of signed numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

-x is the negative value of **x**.

Quotient & Remainder

Computes the integer quotient and the remainder of the inputs.



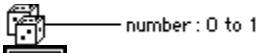
x and **y** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x-y*floor(x/y) is the remainder, which corresponds to the modulo function of other programming languages. When **y** is 1, the remainder is the fractional part of **x**.

floor(x/y) is the integer quotient. If either input is a floating-point number, the quotient is a floating-point number with an integer value. When **y** is 1, the quotient is the integer part of **x**. **floor(x/y)** always rounds to -. For example, if x is -5 and y is 2, the quotient is -3 and the remainder is 1.

Random Number (0-1)

Produces a double-precision floating-point number between 0 and 1, exclusively. The distribution is uniform.



number: 0 to 1 is a double-precision floating-point number between 0 and 1.

Reciprocal

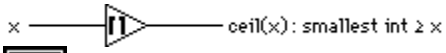
Divides 1 by the input value.

x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

1/x is infinity if **x** is 0. If **x** is an integer, **1/x** is a double-precision floating-point number.

Round To +Infinity

Rounds the input to the next highest integer. For example, if the input is 3.1, the result is 4. If the input is -3.1, the result is -3.

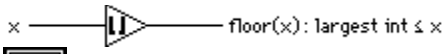


x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

ceil(x): smallest int x.

Round To -Infinity

Truncates the input to the next lowest integer. For example, if the input is 3.8, the result is 3. If the input is -3.8, the result is -4.

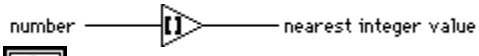


x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

floor(x): largest int x.

Round To Nearest

Rounds the input to the nearest integer. If the value of the input is midway between two integers (for example, 1.5 or 2.5), the function returns the nearest even integer (2).



number can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

nearest integer value.

Scale By Power Of 2

Multiplies one input (**x**) by 2 raised to the power of the other input (**n**). If **n** is floating-point, this function rounds **n** prior to scaling **x** (0.5 rounds to 0; 0.51 rounds to 1). If **x** is an integer, this function is the equivalent of an arithmetic shift.

n and **x** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x*2^n.

Sign

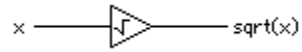
Returns 1 if the input value is greater than 0, returns 0 if the input value is equal to 0, and returns -1 if the input value is less than 0. Other programming languages typically call this function the `signum` or `sgn` function.

number can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

-1, 0, 1.

Square Root

Computes the square root of the input value.

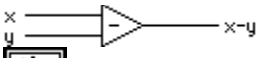


x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

sqrt(x) is a double-precision floating-point number if **x** is an integer. If **x** is less than 0, **sqrt(x)** is not a number (NaN), unless **x** is complex.

Subtract

Computes the difference of the inputs.



x and **y** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x-y.

Polymorphism for Numeric Functions

Click here to access the Numeric Function Descriptions topic.

The arithmetic functions take numeric input data. With some exceptions noted in the function descriptions, the output has the same numeric representation as the input or, if the inputs have different representations, the output is the wider of the inputs.

The arithmetic functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows.

> *Numeric type* = numeric scalar || array [*numeric type*] || cluster [*numeric types*]
>
> The numeric scalars can be floating-point, integers or complex, floating-point numbers. LabVIEW does not allow you to use arrays of arrays.

Arrays can have any number of dimensions of any size. Clusters can have any number of elements. The output type of functions is of the same numeric representation as the input type. For functions with one input, the functions operate on each element of the structure.

For functions with two inputs, you can use the following input combinations.

- *Similar*--both inputs have the same structure, and the output has the same structure as the inputs.
- *One scalar*--one input is a numeric scalar, the other is an array or cluster, and the output is an array or cluster.
- *Array of*--one input is a numeric array, the other is the numeric type itself, and the output is an array.

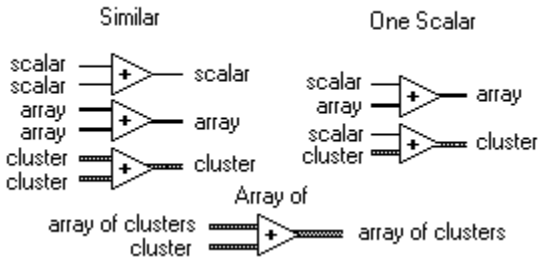
For similar inputs, LabVIEW performs the function on the respective elements of the structures. For example, LabVIEW can add two arrays element-by-element. Both arrays must have the same dimensionality. You can add arrays with differing numbers of elements; the output of such an addition has the same number of elements as the smallest input. Both clusters must have the same number of elements, and the respective elements must have the same structure.

Note: You cannot use the multiply function to do matrix multiplication. If you use the multiply function with two matrices, LabVIEW takes the first number in the first row of the matrix, multiplies it by the first number in the first row of the matrix, and so on. Click here for a description of functions for Linear Algebra.

For operations involving a scalar and an array or cluster, LabVIEW performs the function on the scalar and the respective elements of the structure. For example, LabVIEW can subtract a number from all elements of an array, regardless of the dimensionality of the array.

For operations that involve a numeric type and an array of that type, LabVIEW performs the function on each array element. For example, a graph is an array of points, and a point is a cluster of two numeric types, *x* and *y*. To offset a graph by 5 units in the *x* direction and 8 units in the *y* direction, you can add a point, (5, 8), to the graph.

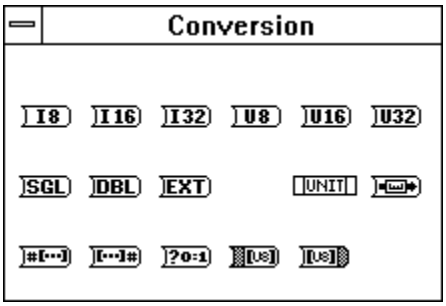
The following figure illustrates some of the possible polymorphic combinations of the Add function.


Similar
scalar
scalar
scalar
array
array
array
cluster
cluster
cluster
One Scalar
scalar
array
array
scalar
cluster
cluster
Array of
array of clusters
cluster
array of clusters

Conversion Function Descriptions

Click here to access the Conversion Function Overview topic.

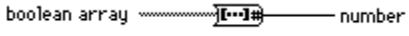
The following illustration shows the options that are available on the **Conversion** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Boolean Array To Number
Boolean To (0,1)
Byte Array To String
Cast Unit Bases
Convert Unit
Number To Boolean Array
String To Byte Array
To Byte Integer
To Double Precision Float
To Extended Precision Float
To Long Integer
To Single Precision Float
To Unsigned Byte Integer
To Unsigned Long Integer
To Unsigned Word Integer
To Word Integer

Boolean Array To Number

Converts **boolean array** to an unsigned byte, word, or long integer by interpreting it as the twos complement representation of an integer with the 0th element of the array being the least significant bit.

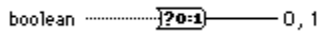


boolean array can have any number of dimensions. This function truncates the **boolean array** if it is too long and pads it with Boolean FALSE bits if the **boolean array** is too short.

number is an unsigned byte, word, or long integer. See the Polymorphism for Boolean Functions topic for more information.

Boolean To (0,1)

Converts a Boolean value to a word integer0 and 1 for the input values FALSE and TRUE, respectively.



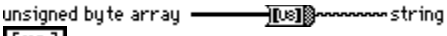
Boolean can be a scalar, an array, or a cluster of Boolean values, an array of clusters of Boolean values, and so on. See the Polymorphism for Boolean Functions topic for more information.

0, 1. The output is 0 if **boolean** is FALSE, 1 if **boolean** is TRUE. **0, 1** is of the same data

structure as **boolean**.

Byte Array To String

Converts an array of unsigned bytes into a **string**..

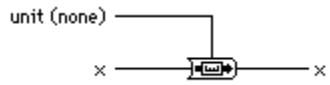


unsigned byte array.

string is the function that obtains each character of the **string** by interpreting each array element as an ASCII value. Refer to ASCII Codes, for the numbers that correspond to each character.

Cast Unit Bases

Changes the units associated with the input to the units associated with **unit** and returns the results at the output terminal. Use this function with extreme care. Because this function works with bases, you must understand the conversion from an arbitrary unit to its bases before you can effectively use this function. Use this function with extreme care. This function can change base units, such as changing meters to grams.



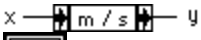
unit (none) associates input with output results, if wired. If left unwired, **unit (none)** does not associate any units.

x.

x returns the same data structure as **x**.

Convert Unit

Converts a physical number (a number that has a unit) to a pure number (a number with no units) or a pure number to a physical number.



X is a floating-point number with or without a unit.

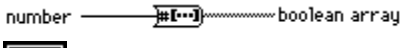
Y is a floating-point number. **Y** has a unit if **X** does not have a unit; **Y** does not have a unit if **X** does have a unit.
You can edit the string inside of the unit by placing the Operating tool on the string and double-clicking, or by highlighting the string with the Labeling tool and then entering the text.

If the input is a pure number, the output receives the specified units. Thus, given an input of 13 and a unit specification of seconds(s), the resulting value is 13 seconds.

If the input is a physical number, and **unit** is a compatible unit, the output is the input measured in the specified units. Thus, if you specify 37 meters(m), and a **unit** is m, the result is 37 with no associated units. If **unit** is feet (ft), the result is 121.36 with no associated units.

Number To Boolean Array

Converts **number** to a Boolean array of 8, 16, or 32 elements, where the 0th element corresponds to the least significant bit (LSB) of the twos complement representation of the integer.



number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. If **number** is a floating point, the function rounds it to a long integer before converting it into a **boolean array**. See Polymorphism for Boolean Functions topic for more information.

boolean array can have 8, 16, or 32 elements.

String To Byte Array

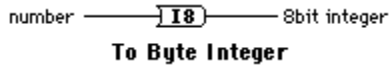
Converts a **string** into an array of unsigned bytes.

string is the input string the function searches.

unsigned byte array. The 0th byte in the array has the ASCII value of the first character in **string**, the first byte has the second value, and so on.

To Byte Integer

Converts **number** to an 8-bit integer in the range -128 to 127.



number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

8bit integer is of the same data structure as **number**.

To Double Precision Float

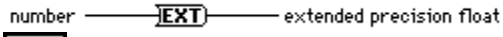
Converts **number** to a double-precision floating-point number.

number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

double precision float is of the same data structure as **number**.

To Extended Precision Float

Converts **number** to an extended-precision floating-point number.



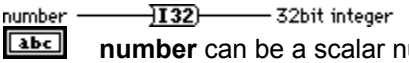
number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

extended precision float is of the same data structure as **number**.

To Long Integer

Converts **number** to a 32-bit integer in the range -2^{31} to

$+2^{31} - 1$.

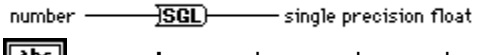


number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

32bit integer is of the same data structure as **number**.

To Single Precision Float

Converts **number** to a single-precision floating-point number.



number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

single precision float is of the same data structure as **number**.

To Unsigned Byte Integer

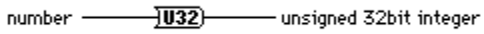
Converts **number** to an 8-bit unsigned integer in the range 0 to 255.

number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

unsigned 8bit integer is of the same data structure as **number**.

To Unsigned Long Integer

Converts **number** to a 32-bit unsigned integer in the range 0 to 232 -1.

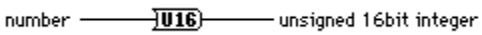


number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

unsigned 32bit integer is of the same data structure as **number**.

To Unsigned Word Integer

Converts **number** to a 16-bit unsigned integer in the range 0 to 65,535.

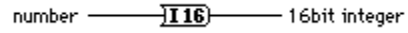


number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

unsigned 16bit integer is of the same data structure as **number**.

To Word Integer

Converts **number** to a 16-bit integer in the range -32,768 to 32,767.



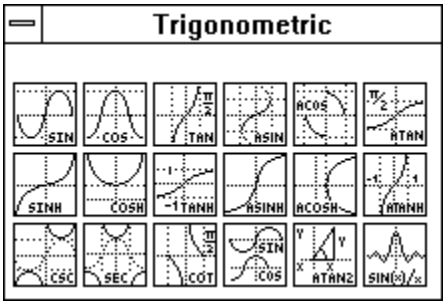
number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Conversion Functions topic for more information.

16bit integer is of the same data structure as **number**.

Trigonometric Function Descriptions

Click here to access the Polymorphism for Trigonometric Functions topic.

The following illustration shows the options for the **Trigonometric** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Cosecant
Cosine
Cotangent
Hyperbolic Cosine
Hyperbolic Sine
Hyperbolic Tangent
Inverse Cosine
Inverse Hyperbolic Sine
Inverse Hyperbolic Cosine
Inverse Hyperbolic Tangent
Inverse Sine
Inverse Tangent
Inverse Tangent (2 Input)
Secant
Sinc
Sine
Sine & Cosine
Tangent

Cosecant

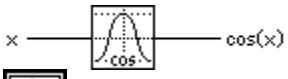
Computes the cosecant of **x**, where **x** is in radians. Cosecant is the reciprocal of sine.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

1/sin(x) is of the same numeric representation as **x**.

Cosine

Computes the cosine of **x**, where **x** is in radians.



x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

cos(x) is of the same numeric representation as **x**.

Cotangent

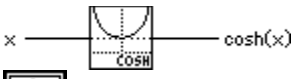
Computes the cotangent of **x**, where **x** is in radians. Cotangent is the reciprocal of tangent.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

1/tan(x) is of the same numeric representation as **x**.

Hyperbolic Cosine

Computes the hyperbolic cosine of **x**.



x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

cosh(x) is of the same numeric representation as **x**.

Hyperbolic Sine

Computes the hyperbolic sine of **x**.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

sinh(x) is of the same numeric representation as **x**.

Hyperbolic Tangent

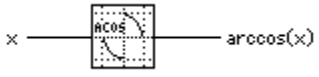
Computes the hyperbolic tangent of **x**.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

tanh(x) is of the same numeric representation as **x**.

Inverse Cosine

Computes the arccosine of **x** in radians. If **x** is not complex and is less than -1 or greater than +1, the result is NaN.

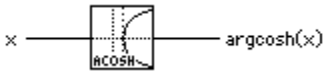


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

arccos(x) is of the same numeric representation as **x**.

Inverse Hyperbolic Cosine

Computes the hyperbolic argcosine of **x**. If **x** is not complex and is less than 1, the result is NaN.

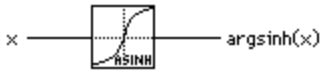


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

argcosh(x) is of the same numeric representation as **x**.

Inverse Hyperbolic Sine

Computes the hyperbolic argsine of **x**.



x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

argsinh(x) is of the same numeric representation as **x**.

Inverse Hyperbolic Tangent

Computes the hyperbolic argtangent of **x**. If **x** is not complex and is less than -1 or greater than 1, the result is NaN.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

argtanh(x) is of the same numeric representation as **x**.

Inverse Sine

Computes the arcsine of **x** in radians. If **x** is not complex and is less than -1 or greater than +1, the result is NaN.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

arcsin(x) is of the same numeric representation as **x**.

Inverse Tangent

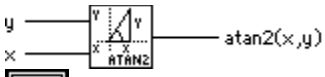
Computes the arctangent of **x** in radians.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

arctan(x) is of the same numeric representation as **x**.

Inverse Tangent (2 Input)

Computes the arctangent of **y**/**x** (which can be between -¹ and ¹) in radians. Thus, this function can compute the arctangent for angles in any of the four quadrants of the xy plane, whereas the Inverse Tangent function computes the arctangent in only two quadrants.



y and **x** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input functions in the Polymorphsim for Numeric Functions topic.

atan2(x,y).

Secant

Computes the secant of **x**, where **x** is in radians. Secant is the reciprocal of cosine.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

1/cos(x) is of the same numeric representation as **x**.

Sinc

Computes the sine of **x** divided by **x**, where **x** is in radians.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

sin(x)/x is of the same numeric representation as **x**.

Sine

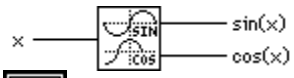
Computes the sine of **x**, where **x** is in radians.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic for more information.

sin(x) is of the same numeric representation as **x**.

Sine & Cosine

Computes both the sine and cosine of **x**, where **x** is in radians. Use this function only when you need both results.

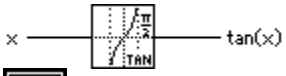


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

sin(x) and **cos(x)** are of the same numeric representation as **x**.

Tangent

Computes the tangent of **x**, where **x** is in radians.



x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphsim for Trigonometric Functions topic.

tan(x) is of the same numeric representation as **x**.

Polymorphism for Trig Functions

Click here to access the Trigonometric Function Descriptions topic.

The trigonometric functions take numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows.

> *Numeric type* = numeric scalar || array [*numeric type*] || cluster [*numeric types*] except that arrays of arrays are not allowed.

Arrays can be any size and can have any number of dimensions. Clusters can have any number of elements. The output type is of the same numeric representation as the input, and the functions operate on each element of the cluster or array. The following list gives the allowable input type combinations for the two-input trigonometric functions. For more information, refer to the discussion of two-input polymorphic functions in the Polymorphism for Numeric Functions topic.

- *Similar*--both inputs have the same structure, and the output has the same structure as the inputs.

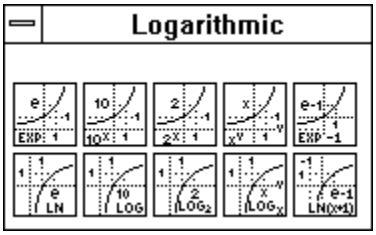
One scalar--one input is a numeric scalar, the other is a numeric array or cluster, and the output is an array or cluster.

Logarithmic Function Descriptions

This topic describes the functions that perform logarithmic operations.

Click here to access the Polymorphism for Logarithmic Functions topic.

The following illustration shows the options for the **Logarithmic** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Exponential
Exponential (Arg) - 1
Power Of 2
Power Of 10
Power Of X
Logarithm Base 2
Logarithm Base 10
Logarithm Base X
Natural Logarithm
Natural Logarithm (Arg + 1)

Exponential

Computes the value of e raised to the **x** power.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

exp(x) is of the same numeric representation as **x**.

Exponential (Arg - 1)

Computes 1 less than the value of e raised to the **x** power. When **x** is very small, this function is more accurate than using the Exponential function and then subtracting 1 from the output.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

exp(x)-1 is of the same numeric representation as **x**.

Logarithm Base 2

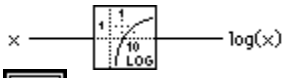
Computes the logarithm of **x** to the base 2. If **x** is 0, **log2(x)** is -. If **x** is not complex and is less than 0, **log2(x)** is NaN.

x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

log2(x) is of the same numeric representation as **x**.

Logarithm Base 10

Computes the logarithm of **x** to the base 10. If **x** is 0, **log(x)** is -. If **x** is not complex and is less than 0, **log(x)** is NaN.

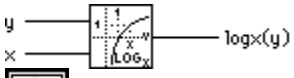


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

log(x) is of the same numeric representation as **x**.

Logarithm Base X

Computes the logarithm of **y** to the base **x** (**x**>0, **y**>0). If **y** is 0, the output is -. When **x** and **y** are both non-complex and **x** is less than or equal to 0, or **y** is less than 0, the output is NaN.



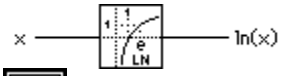
y can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. y must be greater than 0. See the Polymorphism for Logarithmic Functions topic.

x must be greater than 0. **x** can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on.

logx(y).

Natural Logarithm

Computes the natural logarithm of **x**, that is, the logarithm of **x** to the base e. If **x** is 0, **ln(x)** is -. If **x** is not complex and is less than 0, **ln(x)** is NaN.

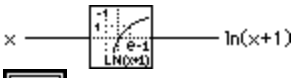


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

ln(x) is of the same numeric representation as **x**.

Natural Logarithm (Arg + 1)

Computes the natural logarithm of (**x** + 1). When **x** is near 0, this function is more accurate than adding 1 to **x** and then using the Natural Logarithm function. If **x** is equal to -1, the result is -. If **x** is not complex and is less than -1, the result is NaN.

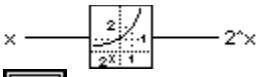


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

ln(x+1) is of the same numeric representation as **x**.

Power Of 2

Computes 2 raised to the **x** power.

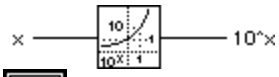


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

2^x is of the same numeric representation as **x**.

Power Of 10

Computes 10 raised to the **x** power.

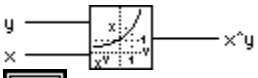


x can be a scalar number, an array or cluster of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Logarithmic Functions topic.

10^x is of the same numeric representation as **x**.

Power Of X

Computes **x** raised to the **y** power. If **x** is not complex, it must be greater than zero unless **y** is an integer value. Otherwise, the result is NaN. If **y** is zero, **x^y** is 1 for all values of **x**, including zero.



y and **x** can be scalar numbers, arrays or clusters of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input functions in Polymorphism for Numeric Functions topic.

x^y.

Polymorphism for Log Functions

Click here to access the Logarithmic Function Descriptions topic.

The logarithmic functions take numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows.

> *Numeric type* = numeric scalar || array [*numeric type*] || cluster [*numeric types*]
>
> except that arrays of arrays are not allowed.

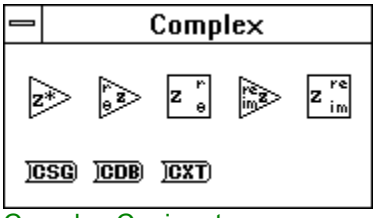
Arrays can be any size and can have any number of dimensions. Clusters can have any number of elements. The output type is of the same numeric representation as the input, and the functions operate on each element of the cluster or array. The following list gives the allowable input type combinations for the two-input logarithmic functions. For more information, refer to the discussion of two-input polymorphic functions in the Polymorphism for Numeric Functions topic.

- *Similar*--both inputs have the same structure, and the output has the same structure as the inputs.
- *One scalar*--one input is a numeric scalar, the other is a numeric array or cluster, and the output is an array or cluster.

Complex Function Descriptions

Click here to access the Polymorphism for Complex Functions topic.

The following illustration displays the options available on the **Complex** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Complex Conjugate
Complex To Polar
Complex To Re/Im
Polar To Complex
Re/Im To Complex
To Double Precision Complex
To Extended Precision Complex
To Single Precision Complex

The functions To Single Precision Complex, To Double Precision Complex, and To Extended Precision Complex convert a numeric input into a specific complex representation. The functions Polar To Complex and Re/Im To Complex create complex numbers from two values given in rectangular or polar notation, and the functions Complex To Polar and Complex To Re/Im break a complex number into its rectangular or polar components.

Complex Conjugate

Produces the complex conjugate of **x + iy**.

x + iy can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on. See the discussion of one-input arithmetic functions in the Polymorphism for Numeric Functions topic for more information.

x - iy.

Complex To Polar

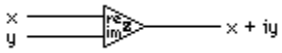
Breaks a complex number into its polar components.

r * e^(i*theta) can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Complex Functions topic for more information.

r and **theta** are of the same data structure as **r * e^(i*theta)**.

Complex To Re/Im

Breaks a complex number into its rectangular components.



x + iy can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Complex Functions topic for more information.

x and **y** are of the same data structure as **x + iy**.

Polar To Complex

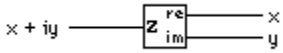
Creates a complex number from two values in polar notation.

r and **theta** can be scalar numbers, clusters of numbers, arrays of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input functions in the Polymorphism for Numeric Functions topic for more information.

r * e^(i*theta) is of the same data structure as **r** and **theta**.

Re/Im To Complex

Creates a complex number from two values in rectangular notation.

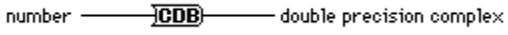


x and **y** can be scalar numbers, clusters of numbers, arrays of numbers, arrays of clusters of numbers, and so on. See the discussion of two-input functions in the Polymorphism for Numeric Functions topic for more information.

x + iy.

To Double Precision Complex

Converts **number** to a double-precision complex number.

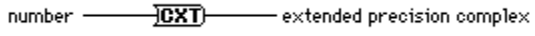


number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Complex Functions topic for more information.

double precision complex is of the same data structure as **number**.

To Extended Precision Complex

Converts **number** to an extended-precision complex number.

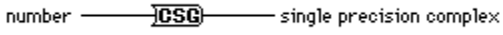


number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Complex Functions topic for more information.

extended precision complex is of the same data structure as **number**.

To Single Precision Complex

Converts **number** to a single-precision complex number.



number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. See the Polymorphism for Complex Functions topic for more information.

single precision complex is of the same data structure as **number**.

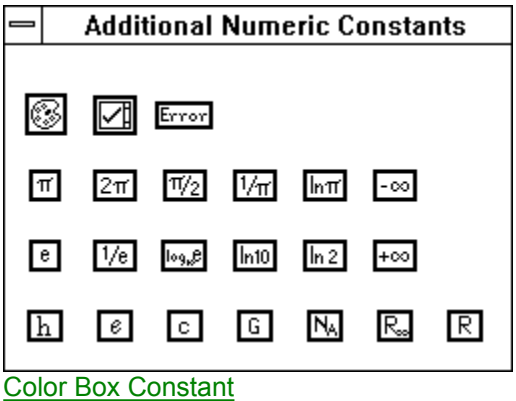
Polymorphism for Complex Functions

Click here to access the Complex Function Descriptions topic.

The complex functions work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Additional Numeric Constants Descriptions

The following illustration displays the options available on the **Additional Numeric Constants** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Color Box Constant
Listbox Symbol Ring
Error Ring
Pi
Pi multiplied by 2
Pi divided by 2
Reciprocal of Pi
Natural Logarithm of Pi
Negative Infinity
Natural Logarithm Base
Reciprocal of e
Base 10 Logarithm of e
Natural Logarithm of 10
Natural Logarithm of 2
Positive Infinity
Planck Constant (J/Hz)
Elementary Charge ©
Speed of Light in Vacuum (m/sec)
Gravitational Constant (N m2/kg2)
Avogadro Constant (1/mol)
Rydberg Constant (/m)
Molar Gas Constant (J/mol K)

Avogadro Constant (1/mol)

Returns the value 6.0220e23.

Base 10 Logarithm of e

Returns the value 0.43429448190325183.

Elementary Charge ©

Returns the value 1.6021892e-19.

Gravitational Constant (N m2/kg2)

Returns the value 6.6720e-11.

Molar Gas Constant (J/mol K)

Returns the value 8.31441.

Natural Logarithm Base

Returns the value 2.71828182845904520.

Natural Logarithm of Pi

Returns the value 1.14472988584940020.

Natural Logarithm of 2

Returns the value 0.69314718055994531.

Natural Logarithm of 10

Returns the value 2.30234095236904570.

Negative Infinity

Returns the value -∞.

Pi

Returns the value 3.14159265358979320.

Pi divided by 2

Returns the value 1.57079632679489660.

Pi multiplied by 2

Returns the value 6.28318530717958650.

Plancks Constant (J/Hz)

Returns the value 6.6262e-34.

Positive Infinity

Returns the value +  .

Reciprocal of e

Returns the value 0.36787944117144232.

Reciprocal of Pi

Returns the value 0.31830988618379067.

Rydberg Constant (/m)

Returns the value 1.097373177e7.

Speed of Light in Vacuum (m/sec)

Returns the value 299,792,458.

Absolute Value Function

Absolute Value

Add Function

Add

Add Array Elements Function

Add Array Elements

Compound Arithmetic Function

Compound Arithmetic

Decrement Function

Decrement

Divide Function

Divide

Increment Function

Increment

Multiply Function

Multiply

Multiply Array Elements Function

Multiply Array Elements

Negate Function

Negate

Quotient & Remainder Function

Quotient & Remainder

Random Number (0-1) Function

Random Number (0-1)

Reciprocal Function

Reciprocal

Round To +Infinity Function

Round To +Infinity

Round To -Infinity Function

Round To -Infinity

Round To Nearest Function

Round To Nearest

Scale By Power Of 2 Function

Scale By Power Of 2

Sign Function

Sign

Square Root Function

Square Root

Subtract Function

Subtract

Conversion Subpalette

Conversion Functions

Trigonometric Subpalette

Trigonometric Functions

Logarithmic Subpalette

Logarithmic Functions

Complex Subpalette

Complex Functions

Boolean Array To Number Function

Boolean Array To Number

Boolean To (0, 1) Function

Boolean To (0,1)

Byte Array To String Function

Byte Array To String

Cast Unit Bases Function

Cast Unit Bases

Convert Unit Function

Convert Unit

Number To Boolean Array Function

Number To Boolean Array

String To Byte Array Function

String To Byte Array

To Byte Integer Function

To Byte Integer

To Double Precision Float Function

To Double Precision Float

To Extended Precision Float Function

To Extended Precision Float

To Long Integer Function

To Long Integer

To Single Precision Float Function

To Single Precision Float

To Unsigned Byte Integer Function

To Unsigned Byte Integer

To Unsigned Long Integer Function

To Unsigned Long Integer

To Unsigned Word Integer Function

To Unsigned Word Integer

To Word Integer Function

To Word Integer

Cosecant Function

Cosecant

Cosine Function

Cosine

Cotangent Function

Cotangent

Hyperbolic Cosine Function

Hyperbolic Cosine

Hyperbolic Sine Function

Hyperbolic Sine

Hyperbolic Tangent Function

Hyperbolic Tangent

Inverse Cosine Function

Inverse Cosine

Inverse Hyperbolic Cosine Function

Inverse Hyperbolic Cosine

Inverse Hyperbolic Sine Function

Inverse Hyperbolic Sine

Inverse Hyperbolic Tangent Function

Inverse Hyperbolic Tangent

Inverse Sine Function

Inverse Sine

Inverse Tangent Function

Inverse Tangent

Inverse Tangent (2 Input) Function

Inverse Tangent (2 Input)

Secant Function

Secant

Sinc Function

Sinc

Sine Function

Sine

Sine & Cosine Function

Sine & Cosine

Tangent Function

Tangent

Exponential Function

Exponential

Exponential (Arg) - 1 Function

Exponential (Arg) - 1

Logarithm Base 2 Function

Logarithm Base 2

Logarithm Base 10 Function

Logarithm Base 10

Logarithm Base X Function

Logarithm Base X

Natural Logarithm Function

Natural Logarithm

Natural Logarithm (Arg + 1) Function

Natural Logarithm (Arg + 1)

Power Of 2 Function

Power Of 2

Power Of 10 Function

Power Of 10

Power Of X Function

Power Of X

Complex Conjugate Function

Complex Conjugate

Complex To Polar Function

Complex To Polar

Complex To Re/Im Function

Complex To Re/Im

Polar To Complex Function

Polar To Complex

Re/Im To Complex Function

Re/Im To Complex

To Double Precision Complex Function

To Double Precision Complex

To Extended Precision Complex Function

To Extended Precision Complex

To Single Precision Complex Function

To Single Precision Complex

Numeric Constant

Use this to supply a constant numeric value to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in a value. You can change the data format and representation. See the Numeric Value Representation Changes for more information.

The value of the numeric constant cannot be changed while the VI executes. You can assign a label to this constant.

Additional Numeric Constants Subpalette

Additional Numeric Constants

Enumerated Constant

Enumerated values associate unsigned integers to strings. If you display a value from an enumerated constant the string is displayed instead of the number associated with it. If you need a set of strings that will not change, then use this constant. Set the value by clicking inside the constant with the Operating Tool and enter the string. To add another item, pop up on the constant and choose **Add Item Before** or **Add Item After**.

The value of the enumerated constant cannot be changed while the VI executes. You can assign a label to this constant.

Ring Constant

Rings can be used to associate unsigned integers to strings. If you display a value from a ring constant the number is displayed instead of the string associated with it. If you need a set of strings that will not change, then use this constant. Set the value by clicking inside the constant with the Operating Tool and enter the string. To add another item, pop up on the constant and choose **Add Item Before** or **Add Item After**.

The value of the ring constant cannot be changed while the VI executes. You can assign a label to this constant.

Listbox Symbol Ring Constant

This ring constant assigns symbols to items in a listbox control. Typically, you wire this constant into the Item Symbols attribute.

Color Box Constant

Use this to supply a constant color value to the block diagram. Set this value by clicking on the constant with the Operating tool and choosing the desired color.

The value of the color box constant cannot be changed while the VI executes. You can assign a label to this constant.

Error Ring Constant

This constant is a predefined ring of errors specific to memory usage, networking, printing, and file I/O. Errors related to DAQ, GPIB, VISA, and Serial VIs and functions are not options in this ring.

Conversion Function Overview

Click here to access the Conversion Function Descriptions topic.

Conversion Functions, General Behavior
Polymorphism for Conversion Functions

Conversion Functions, General Behavior

The following functions convert a numeric input into a specific representation.

- To Byte Integer
- To Double Precision Float
- To Extended Precision Float
- To Long Integer
- To Single Precision Float
- To Unsigned Byte Integer
- To Unsigned Word Integer
- To Unsigned Long Integer
- To Word Integer

When these functions convert a floating-point number to an integer, they round the output to the nearest integer, or the nearest even integer if the fractional part is 0.5. If the result is out of range for the integer, these functions return the minimum or maximum value for the integer type. When these functions convert an integer to a smaller integer, they copy the least significant bits without checking for overflow. When they convert an integer to a larger integer, they extend the sign of a signed integer and pad an unsigned integer with zeros.

Use caution when you convert numbers to smaller representations, particularly when converting integers, because the LabVIEW conversion routines do not check for overflow.

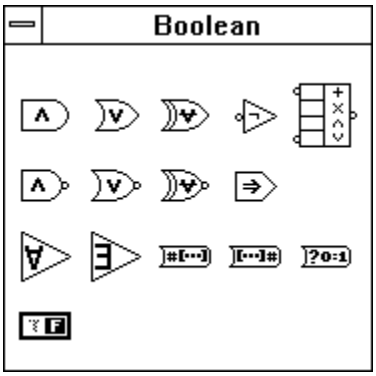
Polymorphism for Conversion Functions

All the conversion functions except Byte Array to String, String to Byte Array, Convert Unit, and Cast Unit Bases are polymorphic. That is, the polymorphic functions work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same numeric representation as the input but with the new type.

Boolean Function Descriptions

This topic describes the functions that perform logical operations. Click here to access the Polymorphism for Boolean Functions topic.

The following illustration shows the **Boolean** palette, which you access by selecting **Functions»Boolean**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

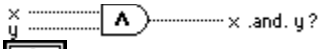


And
And Array Elements
Boolean Array To Number
Boolean Constant
Boolean To (0,1)
Compound Arithmetic
Exclusive Or
Implies
Not
Not And
Not Exclusive Or
Not Or
Number To Boolean Array
Or
Or Array Elements

For examples of some of the arithmetic functions, see `examples\general\structs.llb`.

And

Computes the logical AND of the inputs.



x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Boolean values, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

x .and. y?.

Note: This function performs bit-wise operations on numeric inputs.

And Array Elements

Returns TRUE if all the elements in **boolean array** are true; otherwise it returns FALSE.

boolean array can have any number of dimensions.

logical AND.

Boolean Array To Number

Converts **boolean array** to an unsigned byte, word, or long integer by interpreting it as the two's complement representation of an integer with the 0th element of the array being the least significant bit.

boolean array can have any number of dimensions. This function truncates the **boolean array** if it is too long and pads it with Boolean FALSE bits if the **boolean array** is too short.

number is an unsigned byte, word, or long integer.

Boolean To (0,1)

Converts a Boolean value to a word integer--0 and 1 for the input values FALSE and TRUE, respectively.

boolean can be a scalar, an array, or a cluster of Boolean values, an array of clusters of Boolean values, and so on. See the Polymorphism for Boolean Functions topic for more information.

0, 1. The output is 0 if **boolean** is FALSE, 1 if **boolean** is TRUE. **0, 1** is of the same data structure as **boolean**.

Compound Arithmetic

Performs arithmetic on two or more numeric, cluster, or boolean inputs.

value 0...n can be a number, array of numbers, a cluster, array of clusters, or cluster of booleans. See Polymorphism for Boolean Functions for more information.

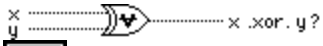
sum, product, AND, or OR of values returns the sum, product, AND, or OR value inputs. You select the operation (multiply, AND, or OR) by popping up on the function and selecting **Change Mode**.

You can invert the inputs or the output of this function by popping up on the individual terminals, and selecting **Invert**. For add, select **Invert** to negate an input or the output. For multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For AND or OR, select **Invert** to logically negate an input or the output.

Note: You add inputs to this node by popping up on an input and selecting Add Input or by placing the Positioning tool in the lower left or right corner of the node and dragging it.

Exclusive Or

Computes the logical Exclusive OR of the inputs.

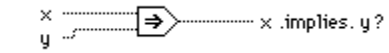


x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

x .xor. y?.

Implies

Computes the logical OR of **y** and of the logical negation of **x**. That is, the function negates **x** and then computes the logical OR of **y** and of the negated **x**.



x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

x .implies. y?.

Not

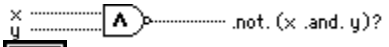
Computes the logical negation of the input.

x can be a scalar Boolean or number, array or cluster of Booleans or numbers, array of clusters of Booleans or numbers, and so on. See the Polymorphism for Boolean Functions topic for more information.

.not. x?.

Not And

Computes the logical NAND of the inputs.

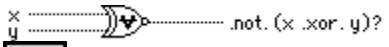


x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

.not. (x .and. y)?.

Not Exclusive Or

Computes the logical negation of the logical exclusive OR of the inputs.

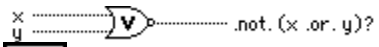


x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

.not. (x .xor. y)?.

Not Or

Computes the logical NOR of the inputs.



x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

.not. (x .or. y)?.

Number To Boolean Array

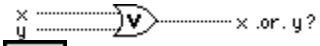
Converts **number** to a Boolean array of 8, 16, or 32 elements, where the 0th element corresponds to the least significant bit (LSB) of the two's complement representation of the integer.

number can be a scalar number, a cluster of numbers, an array of numbers, an array of clusters of numbers, and so on. If **number** is a floating point, the function rounds it to a long integer before converting it into a **boolean array**. See Polymorphism for Boolean Functions for more information.

Boolean array can have 8, 16, or 32 elements.

Or

Computes the logical OR of the inputs.

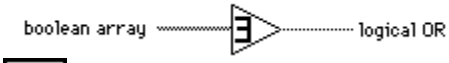


x and **y** must both be Boolean values, or they must both be numbers. They can be scalars, arrays or clusters of numbers or Booleans, arrays of clusters of numbers or Booleans, and so on. See the Polymorphism for Boolean Functions topic for more information.

x .or. y?.

Or Array Elements

Returns FALSE if all the elements in **boolean array** are false; otherwise it returns TRUE.



Boolean array can have any number of dimensions.

logical OR.

And Function

And

And Array Elements Function

And Array Elements

Boolean Array To Number Function

Boolean Array To Number

Boolean To (0,1) Function

Boolean To (0,1)

Compound Arithmetic Function

Compound Arithmetic

Exclusive Or Function

Exclusive Or

Implies Function

Implies

Not Function

Not

Not And Function

Not And

Not Exclusive Or Function

Not Exclusive Or

Not Or Function

Not Or

Number To Boolean Array Function

Number To Boolean Array

Or Function

Or

Or Array Elements Function

Or Array Elements

Boolean Constant

Use this to supply a constant true/false value to the block diagram. Set this value by clicking on the **T** or **F** portion of the constant with the Operating tool. This value cannot be changed while the VI executes.

You can assign a label to this constant.

Polymorphism for Boolean Functions

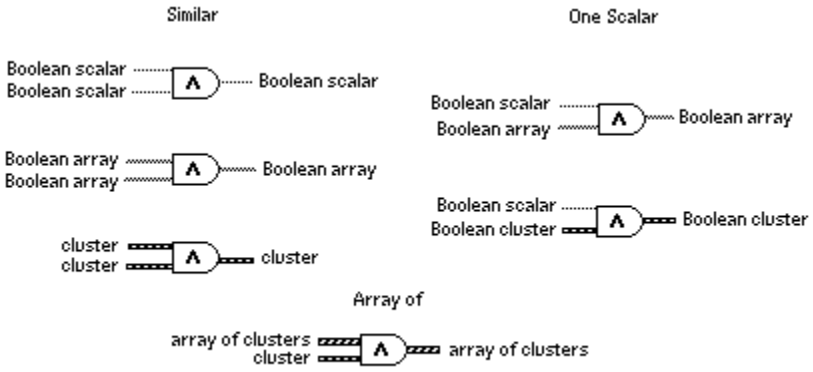
Click here to access the Boolean Function Descriptions topic.

The logical functions take either Boolean or numeric input data. If the input is numeric, LabVIEW performs a bit-wise operation. If the input is an integer, the output has the same representation. If the input is a floating-point number, LabVIEW rounds it to a long integer, and the output is long integer.

The logical functions work on arrays of numbers or Boolean values, clusters of numbers or Boolean values, arrays of clusters of numbers or Boolean values, and so on. A formal and recursive definition of the allowable input type is as follows.

> *Logical type* = Boolean scalar || numeric scalar || array [*logical type*] || cluster [*logical types*]
>
> except that complex numbers and arrays of arrays are not allowed.

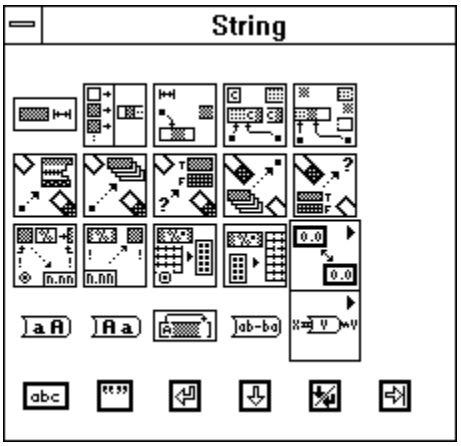
Logical functions with two inputs can have the same input combinations as the arithmetic functions. However, the logical functions have the further restriction that the base operations can only be between two Boolean values or two numbers. For example, you cannot have an AND between a Boolean value and a number. The following illustration shows some combinations of Boolean values for the AND function.



String Function Descriptions

This topic describes the string functions, including those that convert strings to numbers and numbers to strings. Click here to access the String Function Overview topic.

The following illustration shows the **String** palette, which you access by selecting **Functions»String**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



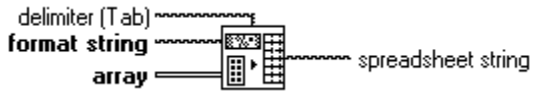
Array To Spreadsheet String
Carriage Return
Concatenate Strings
Empty String
End of Line
Format Into String
Index & Append
Index & Strip
Line Feed
Match Pattern
Pick Line & Append
Reverse String
Rotate String
Select & Append
Select & Strip
Scan From String
Split String
Spreadsheet String To Array
String Constant
String Length
String Subset
Tab
To Lower Case
To Upper Case

Subpalettes

Additional String to Number Functions
String Conversion Functions

Array To Spreadsheet String

Converts an **array** of any dimension to **spreadsheet string**. **spreadsheet string** is a table in string form, containing tabs separating column elements, a platform-dependent EOL character separating rows, and for arrays of three or more dimensions, pages are separated as described below.



format string specifies how to convert the input arguments into **spreadsheet string**. You can use the percent character followed by an s (%s) to convert an **array** of strings to **spreadsheet string**. You can use %d or %f to convert arrays of numbers to **spreadsheet string**. For a complete description of valid **format string** conversion characters, see the Format Strings topic.

array converts each element in **array** according to **format string** and then appends them to **spreadsheet string**.

delimiter (Tab). This function contains an input string, which you can use to specify a character or a string of characters as a delimiter, such as tabs, commas, and so on, to use in your spreadsheet.

spreadsheet string.

For arrays of three-dimensions or more, each page is preceded by a series of indices of the following format.

[*n*,*m*,...,0,0]↵

where

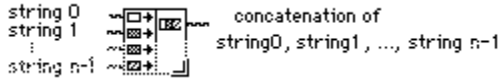
n is the highest dimension index
m is the next highest dimension index
, separates indices
0,0 refers to the first row and column elements of page *n*,*m*,...
represents the end-of-line character

In the following example, a spreadsheet string of a 4x4x3 array appears as it does when you print it. The period character (.) represents actual values of the latter pages, and the end-of-line character ( ) does not appear in an actual printout.

```
[0,0,0]
        1.3     2.6         5.7
3.9             -4.2            6.5
-5.5            9.3                 3.3
9.6             9.8                 0.4

[1,0,0]
.               .                   .
.               .                   .
.               .                   .
.               .                   .

[2,0,0]
.               .                   .
.               .                   .
.               .                   .
.               .                   .

[3,0,0]
```

Concatenate Strings

Concatenates input strings and one-dimensional arrays of strings into a single, output string. For array inputs, this function concatenates each element of the array.

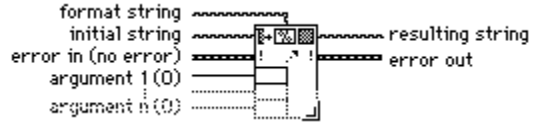


string0 and **string1** are the default input terminals. You can add as many input terminals as you need by selecting **Add Element** from the node pop-up menu, or by resizing the node with the Positioning tool.

concatenation of string0, string1,... contains the concatenated input strings in the order you wire them to the node from top-to-bottom.

Format Into String

Converts input arguments into **resulting string**, whose format is determined by **format string**. You increase the number of parameters by popping up on the node and selecting **Add Parameter** or by placing the Positioning tool over the lower left or right corner of the node and then stretching it until you reach the desired number of arguments.



format string specifies how to convert the input arguments into **resulting string**. If **format string** is not wired, suitable defaults are chosen to match the datatype of the input arguments. See the Format Strings topic for details on format string syntax. If you pop up on Format Into String and select **Edit Format String**, LabVIEW opens a dialog box, which you can use to create and edit your format string. See the *Using String Functions* topic, in Chapter 6, Strings and File I/O, of the *LabVIEW Tutorial Manual* for an example of this feature.

initial string is appended to **resulting string** before processing any arguments.

error in (no error) describes error conditions occurring before this function executes. If an error has already occurred, this function returns the value of the **error in** cluster in **error out**. The function executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this function does not perform any operations.

code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the function that produced the error.

argument 1..n (0) specifies the input parameters to be converted. Each argument may be a string path, enumerated type, or any LabVIEW numeric scalar type. Arrays and clusters cannot be used with the Format Into String function.

resulting string contains the concatenation of **initial string** and the formatted output.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster

contains the same information. Otherwise, **error out** describes the error status of this function.

Format Into String can produce the following errors. If an error occurs, **source** contains a string of the form “Format Into String (arg **n**)”, where **n** is the first argument for which the error occurred.

Error	Code	Description
Format specifier type mismatch	81	The datatype of a format specifier in the format string does not match the datatype of the corresponding input argument.
Unknown format specifier	82	The format string contains an invalid format specifier.
Too few format specifiers	83	There are more arguments than format specifiers.
Too many format specifiers	84	There are more format specifiers than arguments.

Note: If you wire a block diagram constant string to format string, LabVIEW checks for errors in format string at compile time. Such errors must be corrected before you can run the VI. In this case, no errors can occur at run time.

The following table gives examples of format specifiers, which can be used with the Format Into String function. The underline character (_) represent spaces in the output. The last three entries are examples of physical quantity inputs.

format string	argument(s)	resulting string
score= %2d%%	87	score= 87%
level= \n%-7.2e V	0.03642	level= 3.64e-2 V
Name: %s, %s.	Smith John	Name: Smith, John.
Temp: %05.1f %s	96.793 Farenheit	Temp: 096.8 Farenheit
String: %10.5s.	Hello, World	String:_____Hello.
%5.3f	5.67 N	5.670 N
%5.3{mN}f	5.67 N	5670.000 mN
%5.3{kg}f	5.67 N	5.670 ?kg

The last table entry shows the output when the unit in the format specifier is in conflict with the input unit.

Index & Append

Selects a string specified by **index** from **string array** and appends that string to **string**.

string ("") defaults to an empty string if you do not wire it.

index must be scalar. If **index** is a floating-point number, the function rounds it to an integer.

string array. If **string array** finds a match, it returns **string** minus the matched characters in **output string** and the **index** of the string in **string array** that matched. If there is no match, **output string** is **string** and **index** is -1.

output string.

Index & Strip

Compares each string in **string array** with the beginning of **string** until there is a match.

string is the input string the function searches.

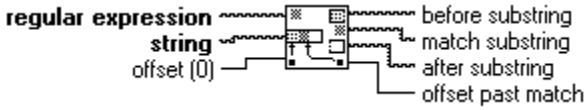
string array. If **string array** finds a match, it returns **string** minus the matched characters in **output string** and the **index** of the string in **string array** that matched. If there is no match, **output string** is **string** and **index** is -1.

index.

output string.

Match Pattern

Searches for **regular expression** in **string** beginning at **offset,** and if it finds a match, splits **string** into three substrings.



regular expression is the pattern searched in **string**. If the function does not find **regular expression**, **match substring** is empty, **before substring** is the entire **string**, **after substring** is empty, and **offset past match** is -1. The tables that follow the input and output descriptions of this function describe special characters for **regular expression**.

string is the input string the function searches.

offset (0) must be scaler. The **offset** of the first character in **string** is 0. If you leave **offset** unwired or its value is less than 0, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

before substring returns a string containing all the characters before the match.

match substring is the matched string.

after substring contains all characters following the matched pattern.

offset past match is the index in **string** of the first character of **after substring**.

The following table describes special characters you can use for regular expression.

Special Character	Interpreted by the Match Pattern Function as...
.	Matches any character.

?

Matches zero or one instances of the expression preceding *?*.

\

Cancels the interpretation of special characters (for example,\? matches a question mark). You can also use the following constructions for the space and nondisplayable characters

\b	backspace
\f	form feed
\n	newline
\s	space
\r	carriage return
xx	any character, where *xx* is the hex code using 0 through 9 and upper case A through F
\t	tab

^

If ^ is the first character of **regular expression**, it anchors the match to the **offset** in **string**. The match fails unless **regular expression** matches that topic of **string** that begins with the character at **offset**. If ^ is not the first character, it is treated as a regular character.

[ ]

Encloses alternates. For example, [abc] matches a, b, or c. The following character has special significance when used within the brackets in the following manner.

- (dash)	Indicate a range when used between digits, or lowercase or uppercase letters (for example, [0-5], [a-g], or [L-Q])

The following characters have significance only when they are the first character within the brackets.

~	Excludes the set of characters, including nondisplayable characters. [~0-9] matches any character other than 0 through 9.
^	Excludes the set with respect to all the displayable characters (and the space characters). [^0-9] gives the space characters and all displayable characters except 0 through 9.

+

Matches the longest number of instances of the expression preceding +; there must be at least one instance to constitute a match.

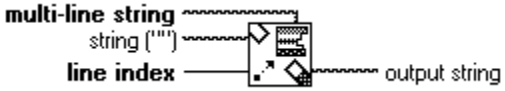
*	Matches the longest number of instances of the expression preceding * in **regular expression**, including zero instances.
$	If $ is the last character of **regular expression**, it anchors the match to the last element of **string**. The match fails unless **regular expression** matches up to and including the last character in the string. If $ is not last, it is treated as a regular character.

The following table gives examples of strings you can wire to the regular expression parameter of the Match Pattern function.

Characters to Be Matched	**regular expression**
VOLTS	VOLTS
All uppercase and lowercase versions of volts, that is, VOLTS, Volts, volts, and so on	[Vv][Oo][Ll][Tt][Ss]
A space, a plus sign, or a minus sign	[+-]
A sequence of one or more digits	[0-9]+
Zero or more Spaces	\s* or * (that is, a space followed by an asterisk)
One or more Spaces, Tabs, Newlines, or Carriage Returns	[\t \r \n \s]+
One or more characters other than digits	[~0-9]+
The word Level only if it begins at the offset position in the string	^Level
The word Volts only if it appears at the end of the string	Volts$
The longest string within parentheses	(.*)
The longest string within parentheses but not containing any parentheses within it	([~( )]*)
The character, [	[[ ]

Pick Line & Append

Chooses a line from **multi-line string** and appends that line to **string**.



string ("") defaults to an empty string if you do not wire it.

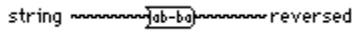
multi-line string contains one or more substrings separated by carriage returns.

line index selects the line the function appends from **multi-line string**. **line index** must be scalar. A **line index** of 0 selects the first line. If **line index** is negative or is greater than or equal to the number of lines in **multi-line string**, the function sets **output string** to **string**. If **line index** is a floating-point number, the function rounds it to an integer.

output string.

Reverse String

Produces a string whose characters are in reverse order of those in **string**.



string can be a string, a cluster, an array of strings, or an array of clusters. If the data type of **string** is a number, the function interprets it as ASCII codes for characters. See the Polymorphism for String Functions topic for more information.

reversed has the same structure as **string.**

Rotate String

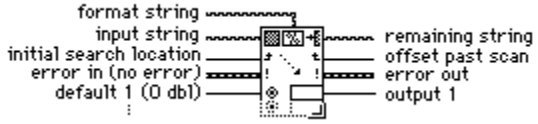
Places the first character of **string** in the last position of **first char last**, shifting the other characters forward one position. For example, the string *abcd* becomes *bcda*.

string can be a string, a cluster, an array of strings, or an array of clusters. If the data type of **string** is a number, the function interprets it as ASCII codes for characters. See the Polymorphism for String Functions topic for more information.

first char last has the same structure as **string.**

Scan From String

Scans the input string and converts the string according to **format string**. You increase the number of parameters by popping up on the node and selecting **Add Parameter** or by placing the Positioning tool over the lower left or right corner of the node and then stretching it until you reach the desired number of parameters.



format string specifies how to convert the input string into the output arguments. See the Format Strings topic for details on format string syntax. If **format string** is unwired, the string is scanned according to default behavior for the datatypes of the outputs wired. If you pop up on Scan From String and select **Edit Format String**, LabVIEW opens a dialog box, which you can use to create and edit your format string.

input string is the string to be scanned.

initial search location is the offset into the string where the scan begins. If left unwired, **initial search location** defaults to 0.

error in (no error) describes error conditions occurring before this function executes. If an error has already occurred, this function returns the value of the **error in** cluster in **error out**. The function executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this function does not perform any

operations.

code is the error code number identifying an error. A value of 0 generally means no error, a negative value means a fatal error, and a positive value is a warning.

source identifies where an error occurred. The source string is usually the name of the VI or function that produced the error.

default 1..n (0 dbl) specifies the default for the output parameters. If the input value cannot be scanned from the string, the default value is used. If the default is not wired, the output is set to zero or the empty string, depending on its datatype.

remaining string returns the portion of the string that remains after scanning all arguments.

offset past scan is the offset of **input string** after completing the scan.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this function.

Scan From String can produce the following errors. If an error occurs, **source** contains a string of the form “Scan From String (arg **n**)“, where **n** is the first argument for which the error occurred.

Error	Code	Description
Format specifier type mismatch	81	The datatype of a format specifier in the format string does not match the datatype of the corresponding output.
Unknown format specifier	82	The format string contains an invalid format specifier.
Too few format specifiers	83	There are more arguments than format specifiers.
Too many format specifiers	84	There are more format specifiers than arguments.
Scan failed	85	Scan From String was unable to convert the input string into the datatype indicated by the format specifier.

Note: If you wire a block diagram constant string to format string, LabVIEW checks for errors in format string at compile time. You must correct these errors before you can run the VI. In this case, only ÒScan failedÓ can occur at run time.

output 1..n specifies the output parameters to write. Each output may be a string path, enumerated type, or any LabVIEW, numeric scalar type. Arrays and clusters cannot be used with the Scan From String function.

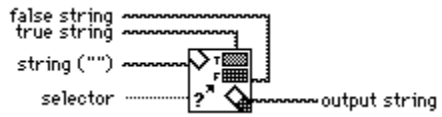
The following table shows examples of how Scan From String operates with various arguments.

input string	format string	default(s)	output(s)	remaining string
abc,xyz 12.3+56i 7200	%s,%s%f%2d		abc xyz 12.3+56i 7	00
Q+1.27E-3 tail	Q%f t		1.27E-3	ail

0123456789	%3d%3d		12 345	6789
X:9.860 Z:3.450	X:%fY:%f	100 (I32) 100.0 (DBL)	10 100.0	Z: 3450
set49.4.2	set%d		49	.4.2

Select & Append

Selects either a false or true string according to a Boolean **selector** and appends that string to **string**.



string (" ") defaults to an empty string if you do not wire it.

false string.

true string.

selector. If **selector** is TRUE, the function appends **true string** to **string**. If **selector** is FALSE, the function appends **false string** to **string**.

output string.

Select & Strip

Examines the beginning of **string** to see whether it matches **true string** or **false string**. This function returns a Boolean TRUE or FALSE value in **selection**, depending on whether **string** matches **true string** or **false string**.

string is the input string the function searches.

true string.

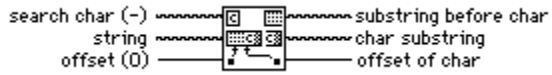
false string.

selection is TRUE if **true string** matches the beginning of **string**, and FALSE if **false string** matches the beginning of **string**, or if neither string matches.

output string. If either **true string** or **false string** matches the beginning of **string**, **output string** is a shortened version of **string** with the matching substring removed. If neither **true string** nor **false string** matches the beginning of **string**, **output string** is **string**.

Split String

Splits the string at offset or searches for the first occurrence of **search char** in the **string**, beginning at **offset**, and splits the string at that point.



search char (-). If the function does not find **search char** in **string**, the function sets **offset of char** to -1, **substring before char** to the entire **string**, and **char substring** to an empty string. If you do not wire **search char**, or if it is an empty string, the function splits **string** at **offset**. You must wire either **search char** or **offset**.

string is the input string the function searches or splits.

offset (0) must be scaler. **offset** is the starting position for the search. The offset of the first character in the **string** is 0. If you leave **offset** unwired, or if its value is less than 0, it defaults to 0. You must wire either **search char** or **offset**. If **offset** is a floating-point number, the function rounds it to an integer.

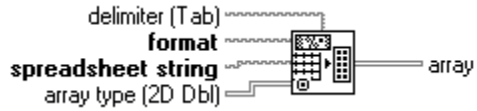
substring before char is all the characters in **string** before **search char**.

char substring consists of **search char** and all subsequent characters in **string**.

offset of char is the position of **search char** in **string**.

Spreadsheet String To Array

Converts the **spreadsheet string** to a numeric **array** of the dimension and representation of **array type**. This function works for arrays of strings as well as arrays of numbers.



format string. You can use the percent character followed by an s (%s) to convert an **array** of strings to the **spreadsheet string**. You can use %d or %f to convert arrays of numbers. For a complete description of valid **format string** conversion characters, see the Format Strings topic.

spreadsheet string. Delimiters, such as tabs, separate columns in **spreadsheet string**, and an End-of-Line (EOL) character separates rows. The function converts each element in **spreadsheet string** according to **format** and then stores them in **array**.

delimiter (Tab) This function uses an input string, which you can use to specify a character or a string of characters as a delimiter, such as tabs, commas, and so on, to use in your spreadsheet.

array type (2D Dbl). If you leave **array type** unwired, it defaults to a 2D array of double-precision, floating-point numbers. See the Array To Spreadsheet String function description in this chapter for information on arrays of three-dimensions or more.

array.

String Length

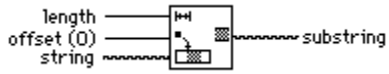
Returns in **length** the number of characters (bytes) in **string**.

string can be a scalar, *n*-dimensional array, or cluster. See the Polymorphism for String Functions topic for more information.

length has the same structure as **string**, but the type of **length** is a long integer.

String Subset

Returns the **substring** of the original **string** beginning at **offset** and containing **length** number of characters.



length must be scalar. If **length** is a floating-point number, the function rounds it to an integer.

offset (0) must be scalar. The **offset** of the first character in the string is 0. If you do not wire **offset**, or if it is less than 0, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

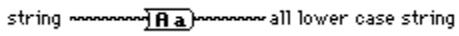
string is the input string the function searches. The length of the string minus the offset is the length of **substring**.

substring is empty if **offset** is greater than the length of the **string** or if the **length** is less than or

equal to 0. If **length** is greater than the length of **string** minus **offset**, **substring** is the remainder of string beginning at offset.

To Lower Case

Converts all alphabetic characters in **string** to lowercase characters. This function does not affect nonalphabetic characters.

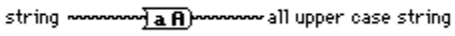


string can be a string, a cluster, an array of strings, or an array of clusters. If the data type of **string** is a number, the function interprets it as ASCII codes for characters. See the Polymorphism for String Functions topic for more information.

all lower case string has the same structure as **string.**

To Upper Case

Converts all alphabetic characters in **string** to uppercase characters. This function does not affect nonalphabetic characters.



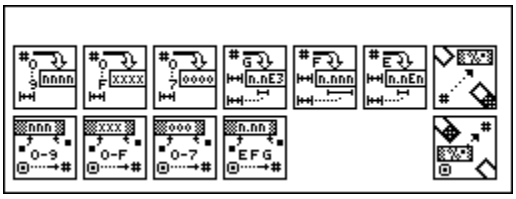
string can be a string, a cluster, an array of strings, or an array of clusters. If the data type of **string** is a number, the function interprets it as ASCII codes for characters. See the Polymorphism for String Functions topic for more information.

all upper case string has the same structure as **string.**

Additional String To Number Function Descriptions

For general information about Additional String to Number functions, see Polymorphism for Additional String to Number Functions.

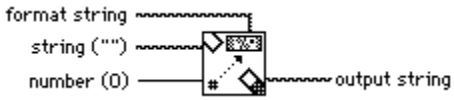
The following illustration displays the options available on the **Additional String to Number Functions** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Format & Append
Format & Strip
From Decimal
From Exponential/Fract/Eng
From Hexadecimal
From Octal
To Decimal
To Engineering
To Exponential
To Fractional
To Hexadecimal
To Octal

Format & Append

Converts **number** into a regular LabVIEW string according to the format specified in **format string,** and appends this to **string**.



Note: The Format Into String function has the same functionality as Format & Append but can use multiple inputs, so that you can convert information simultaneously. You should consider using Format Into String instead of this function: in many cases, this can simplify your block diagram.

format string specifies how to convert **number** into **output string**. See the Format Strings topic for details on format string syntax.

string ("") defaults to an empty string if you do not wire it.

number (0) must be scalar but can be a physical quantity (a value with an associated unit).

output string is the result of converting **number** to a string, and appending this to string.

Format & Strip

Looks for **format string** at the beginning of **string**, formats any number in this string portion according to the conversion codes in **format string**, and returns the converted number in **number** and the remainder of **string** after the match in **output string**.

string is the input string the function searches.

format string specifies how to convert **string** into **output string**. See the Format Strings topic for details on format string syntax.

default (0.0 dbl). You specify the numeric representation for **number** by wiring any object of the appropriate representation to **default**. If you leave **default** unwired, the parameter defaults to a double-precision, floating-point value of 0.

number. If there is no match, **number** is the value in **default**. If **number** is an interger, it may overflow if the input is out-of-range. In that case, **number** is set to maxint or -maxint-1. For example, if the input string is 300, and the data type is int 8, LavVIEW sets the value to 127.

output string. If there is no match, **output string** is **string**.

From Decimal

Converts the numeric characters in **string**, starting at **offset**, to a decimal integer and returns it in **number**.

string can be a string, a cluster, an array of strings, or an array of clusters. **string** can include a plus or minus sign prefix. See the Polymorphism for Additional String to Number Functions topic for more information.

offset must be scalar. If there are no numeric characters at **offset**, the function returns **default**, and **offset past number** is the same as **offset**. If the number the function finds at **offset** is fractional, the function returns only the integer portion in **number**. If you do not wire **offset**, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

default(0L). If you leave **default** unwired, it defaults to a 32-bit integer value of 0.

number can be a number, a cluster, or an array of numbers, depending on the structure of **string** and **offset**. The following table shows how the values of **string**, **offset**, and **default** affect **number**. If **number** is an interger, it may overflow if the input is out-of-range. In that case, **number** is set to maxint or -maxint-1. For example, if the input string is 300, and the data type is int 8, LavVIEW sets the value to 127.

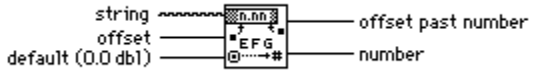
offset past number is the index in **string** of the first character following the number. The following table shows how the values of **string**, **offset**, and **default** affect **offset past number**. **offset past number** reflects the value from the last string if you input an array of strings.

string	offset	default	offset past number	number	Comments
13ax	0	0	2	13	
-4.8bcde conversion	0	0	2	-4	Because an integer is being converted, stops at the decimal point.
a49b	0	-9	0	-9	**default** is used since

no digits were read.

From Exponential/Fract/Eng

Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, or exponential or fractional format and returns it in **number**.



string can be a string, a cluster, an array of strings, or an array of clusters. See the Polymorphism for Additional String to Number Functions for more information.

offset is a scalar number. If there are no numeric characters at **offset**, the function returns **default**, and **offset past number** is the same as **offset**. If you leave **offset** unwired, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

default (0.0 dbl). If you leave **default** unwired, it defaults to 0.0 in double-precision.

number can be a number, a cluster, an array of numbers, or an array of clusters, depending on the structure of **string** and **offset**. The following table shows how the values of **string**, **offset**, and **default** affect **number**.

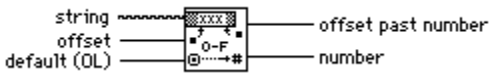
offset past number is the index in **string** of the first character following the number. The following table shows how the values of **string**, **offset**, and **default** affect **offset past number**. **offset past number** reflects the value from the last string if you input an array of strings.

string	offset	default	offset past number	number	Comments
-4.7e-3x	0	0	7	-.0047	x is not allowed.
+5.3.2	0	0	4	5.3	Second decimal point not allowed, so conversion stops there.

Note: If you wire the characters Inf or NaN to string, this function returns the LabVIEW values Inf and NaN, respectively.

From Hexadecimal

Interprets the characters 0 through 9, A through F, and a through f in **string** starting at **offset** as a hex integer and returns it in **number**.



string can be a string, a cluster, an array of strings, or an array of clusters. See the Polymorphism for Additional String to Number Functions for more information.

offset must be scaler. If there are no hex characters at **offset**, the function returns **default**, and

offset past number is the same as **offset**. If you leave **offset** unwired, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

default (0L). If you leave **default** unwired, it defaults to 0 in an unsigned 32-bit integer.

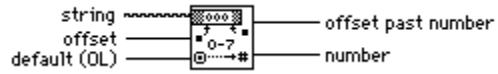
number can be a number, a cluster, an array of numbers, or an array of clusters depending on the structure of **string** and **offset**. The following table shows how the values of **string**, **offset**, and **default** affect **number**. If the input string represents a **number** outside the range of **number**, **number** is set to maxint for that data type.

offset past number is the index in **string** of the first character following the number. The following table shows how the values of **string**, **offset**, and **default** affect **offset past number**. **offset past number** reflects the value from the last string if you input an array of strings.

string	offset	default	offset past number	number	Comments
f3g	0	0	2	243	
-30	0	0	0	0	Negative numbers are not permitted for hex.

From Octal

Interprets the characters 0 through 7 in **string** starting at **offset** as an octal integer and returns it in **number**. This function also returns the index in **string** of the first character following the number.



string can be a string, a cluster, an array of strings, or an array of clusters. See the Polymorphism for Additional String to Number Functions for more information.

offset must be scaler. If there are no octal characters at **offset**, the function returns **default**, and **offset past number** is the same as **offset**. If you leave **offset** unwired, it defaults to 0. If **offset** is a floating-point number, the function rounds it to an integer.

default (0L). If you leave **default** unwired, it defaults to an unsigned 32-bit integer value of 0.

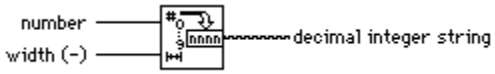
number can be a number, a cluster, an array of numbers, or an array of clusters, depending on the structure of **string** and **offset**. The following table shows how the values of **string**, **offset**, and **default** affect **number**. If the input string represents a **number** outside the range of **number**, **number** is set to maxint for that data type.

offset past number is the index in **string** of the first character following the number. The following table shows how the values of **string**, **offset**, and **default** affect **offset past number**. **offset past number** reflects the value from the last string if you input an array of strings.

string	offset	default	offset past number	number	Comments
92	0	0	0	0	9 is not an octal digit.
071a	0	0	3	57	

To Decimal

Converts **number** to a string of decimal digits **width** characters wide, or wider if necessary.



number can be a scaler number, array or cluster of numbers, array of clusters of numbers, and so on. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

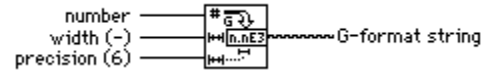
width (-) must be scalar. If you leave **width** unwired, it defaults to 0, which means that **decimal integer string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

decimal integer string. The following table shows how the values of **number** and **width** affect **decimal integer string**. In this table, the underline character (_) represents spaces in **decimal integer string**.

number	width	decimal integer string	Comments
4.6	2	_5	Floating-point numbers are rounded to integers.
3.0	4	_ _ _3	If **width** is larger than needed, spaces are added on the left.
-311	3	-311	If **width** is inadequate, **decimal integer string** is as large as necessary.

To Engineering

Converts **number** to an engineering format, floating-point string **width** characters wide, or wider if necessary. Engineering format is similar to E format, except the exponent is a multiple of three (É, -3, 0, 3, 6, É).



number. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

width (-) must be scalar. If you leave **width** unwired, it defaults to 0, which means that **Engineering string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

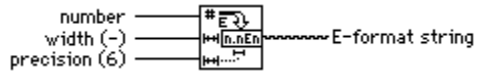
precision (6) must be scalar. The function rounds the number of digits after the decimal point of **Engineering string** to **precision**. If **precision** is 0, **Engineering string** does not contain a decimal point and contains at most three digits of the mantissa. If **precision** is a floating-point number, the function rounds it to an integer. If you leave **precision** unwired, it defaults to 6.

Engineering string. The following table shows how the values of **number**, **width**, and **precision** affect **Engineering string**. In the following table, the underline character (_) represents a space in **Engineering string**.

number	width	precision	Engineering string	Comments
4.93	10	2	_ _ _ _4.93e0	**Number** is rounded, padded with spaces on the left.
.49	10	2	_ _ _ _490e-3	**Number** is rounded, padded with spaces on the left.
61.96	8	1	_ _62.0e0	**Number** is rounded, padded with spaces on the left.
1789.32	8	2	_ _1.79e3	**Number** is rounded, padded with spaces on the left.

To Exponential

Converts **number** to an E-format (exponential notation), floating-point string **width** characters wide, or wider if necessary.



number. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

width (-) must be scalar. If **width** is unwired, it defaults to 0, which means that **E-format string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

precision (6) must be scalar. The function rounds the number of digits after the decimal point of **E-format string** to **precision**. If **precision** is 0, the **E-format string** contains no decimal point and only a single digit of the mantissa. If **precision** is a floating-point number, the function rounds it to an integer. If you leave **precision** unwired, it defaults to 6.

E-format string. The following table shows how the values of **number**, **width**, and **precision** affect **E-format string**. In the following table, the underline character (_) represents a space in **E-format string**.

number	width	precision	E-format string	Comments
4.911	5	2	4.91e0	**Number** is rounded, width is extended.

.003926	10	2	_ _ _3.93e-3	**Number** is rounded, padded with spaces on the left.
216.01	5	0	_ _2e2	**Number** is rounded, padded with spaces on the left.

To Fractional

Converts **number** to an F-format (fractional notation), floating-point string **width** characters wide, or wider if necessary.

number. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

width (-) must be scalar. If you leave **width** unwired, it defaults to 0, which means that **F-format string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

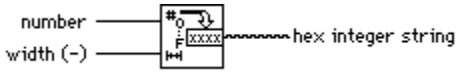
precision (6) must be scalar. The function rounds the number of digits after the decimal point of **F-format string** to **precision**. If **precision** is 0, **F-format string** contains no decimal point or fractional portion. If **precision** is a floating-point number, the function rounds it to an integer. If you leave **precision** unwired, it defaults to 6.

F-format string. The following table shows how the values of **number**, **width**, and **precision** affect **F-format string**. **F-format string** can be Inf, -Inf, or NaN if the value you wire to **number** is infinity or is not a number. In the following table, the underline character (_) represents a space in **F-format string**.

number	width	precision	F-format string	Comments
4.911	6	2	_ _4.91	**Number** is rounded, padded with spaces on the left.
.003926	8	4	_ _0.0039	**Number** is rounded, padded with spaces on the left.
-287.3	5	0	_-287	**Number** is rounded, padded with spaces on the left.

To Hexadecimal

Converts **number** to a string of hexadecimal digits **width** characters wide, or wider if necessary.



number. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

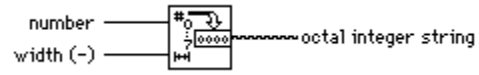
width (-) must be scalar. If you leave **width** unwired, it defaults to 0, which means that **hex integer string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

hex integer string. The following table shows how the values of **number** and **width** affect **hex integer string**.

number	width	hex integer string	Comments
3	4	0003	If **width** is larger than needed, zeros are added on the left.
42	3	02A	
-4.2	3	FFFFFFFC	-4.2 is rounded up to -4 in 32-bit integer format. **width** is too small to represent the hex version of a negative number, so the field width is extended.

To Octal

Converts **number** to a string of octal digits **width** characters wide, or wider if necessary.



number. If **number** is a floating-point number, the function rounds it to the nearest long integer value. See the Polymorphism for Additional String to Number Functions topic for more information.

width (-) must be scalar. If you leave **width** unwired, it defaults to 0, which means that **octal integer string** is as small as possible. If **width** is a floating-point number, the function rounds it to an integer.

octal integer string. The following table shows how the values of **number** and **width** affect **octal integer string**.

number	width	octal integer string	Comments
3	4	0003	
42	3	052	

-4.2	3	37777777774	-4.2 is rounded up to -4 in 32-bit integer format. **width** is too small to represent the octal version of a negative number, so the field width is extended.

Polymorphism for Additional String to Number Functions

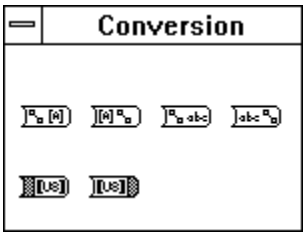
Click here to access the Additional String To Number Functions topic.

To Decimal, To Hex, To Octal, To Engineering, To Fractional, and To Exponential accept clusters and arrays of numbers and produce clusters and arrays of strings. From Decimal, From Hex, From Octal, and From Exponential/Fract/Sci accept clusters and arrays of strings and produce clusters and arrays of numbers. Width and precision inputs must be scalar.

String Conversion Function Descriptions

For general information about String Conversion functions, see Polymorphism for String Conversion Functions .

The following illustration shows the **String Conversion** subpalette Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

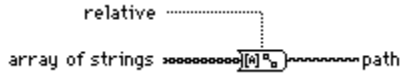


Array Of Strings To Path
Byte Array To String
Path To Array Of Strings
Path To String
Refnum To Path
String To Byte Array
String To Path

Array Of Strings To Pathaccepts one-dimensional (1D) arrays of strings, Path To Array Of Strings accepts paths, Path To Stringaccepts paths, and String To Pathaccepts strings.

Array Of Strings To Path

Converts an **array of strings** into a relative or absolute **path**.



array of strings contains the names of the components of the path you want to build. The first element is the highest level of the path hierarchy (the volume name, for file systems that support multiple volumes), and the last element is the last element of the hierarchy. An element that contains an empty string tells LabVIEW to go up a level in the hierarchy.

relative indicates whether you want to create a **relative** path or an absolute path. If you set **relative** to TRUE, it is a relative path; if set to FALSE, it is an absolute path. If you set **relative** to FALSE, and the path sepcified is not valid as an absolute path (for example, the path means *go up a level*), the function sets **path** to not a path.

path is the resulting path.

Byte Array To String

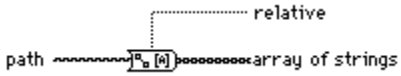
Converts an array of unsigned bytes into a **string**..

unsigned byte array.

string is the function that obtains each character of the **string** by interpreting each array element as an ASCII value. Refer to ASCII Codes for the numbers that correspond to each character.

Path To Array Of Strings

Converts a **path** into an **array of strings** and indicates whether the path is **relative**.



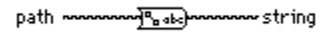
path is the path you want to convert to an array of strings. If **path** is not-a-path, the **array of strings** is empty and **relative** is FALSE.

relative indicates whether you want to create a **relative** path or an absolute path. If you set **relative** to TRUE, it is a relative path; if set to FALSE, it is an absolute path. If you set **relative** to FALSE, and the path sepcified is not valid as an absolute path (for example, the path means *go up a level*), the function sets **path** to not a path.

array of strings contains the components of the path. The first element is the first step of the path hierarchy (the volume name, for file systems that support multiple volumes), and the last element is the file or directory specified by the path.

Path To String

Converts **path** into a string describing a path in the standard format of the platform.

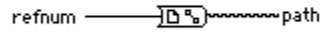


path is the path you want to convert to a string. If **path** is not-a-path, the function sets **string** to not a path.

string is the path description in the standard format for the current platform of **path**. **string** is of the same data structure as **path**.

Refnum To Path

Returns the **path** associated with the specified **refnum.**



refnum is the refnum of an open file whose associated path you want to determine. If **refnum** is not a valid refnum, this function sets **path** to Not A Path, meaning the file is closed.

path is the corresponding path.

String To Byte Array

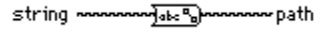
Converts a **string** into an array of unsigned bytes.

string is the input string the function searches.

unsigned byte array. The 0th byte in the array has the ASCII value of the first character in **string**, the first byte has the second value, and so on.

String To Path

Converts a string, describing a path in the standard format for the current platform, to a path.



string can be a string, a cluster of strings, an array of strings, an array of clusters of strings, and so on. See the Polymorphism for String Conversion Functions topic for more information.

path is the platform-dependent representation of the path described by **string**. If **string** is not a valid path description on the current platform, the function sets **path** to not a path. **path** is of the same data structure as **string**.

Polymorphism for String Conversion Functions

The Path To String and String To Path functions are polymorphic. That is, they work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Array To Spreadsheet String Function

Array To Spreadsheet String

Byte Array To String Function

Byte Array To String

Carriage Return Function

Carriage Return

Concatenate Strings Function

Concatenate Strings

Empty String Function

Empty String

End of Line Function

End of Line

Format Into String Function

Format Into String

Index & Append Function

Index & Append

Index & Strip Function

Index & Strip

Line Feed Function

Line Feed

Match Pattern Function

Match Pattern

Pick Line & Append Function

Pick Line & Append

Reverse String Function

Reverse String

Rotate String Function

Rotate String

Scan From String Function

Scan From String

Select & Append Function

Select & Append

Select & Strip Function

Select & Strip

Split String Function

Split String

Spreadsheet String To Array Function

Spreadsheet String To Array

String Length Function

String Length

String Subset Function

String Subset

Tab Function

Tab

To Lower Case Function

To Lower Case

To Upper Case Function

To Upper Case

Alternate String To Number Functions Subpalette

Additional String To Number Functions

Format & Append Function

Format & Append

Format & Strip Function

Format & Strip

From Decimal Function

From Decimal

From Exponential/Fract/Eng Function

From Exponential/Fract/Eng

From Hexadecimal Function

From Hexadecimal

From Octal Function

From Octal

To Decimal Function

To Decimal

To Engineering Function

To Engineering

To Exponential Function

To Exponential

To Fractional Function

To Fractional

To Hexadecimal Function

To Hexadecimal

To Octal Function

To Octal

Array Of Strings To Path Function

Array Of Strings To Path

Path To Array Of Strings Function

Path To Array Of Strings

Path To String Function

Path To String

Refnum to Path Function

Refnum To Path

String To Byte Array Function

String To Byte Array

String To Path Function

String To Path

String Conversion Functions Subpalette

String Conversion Functions

String Constant

Use this to supply a constant ASCII value to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in the value. You can change the display mode so you can see non-displayable characters or the hex equivalent to the characters. You can also set the constant in password display mode so “*” are displayed when you type in characters.

The value of the string constant cannot be changed while the VI executes. You can assign a label to this constant.

String Function Overview

Click here to access the String Function Descriptions topic.

Format Strings
Polymorphism for String Functions

Format Strings

Many LabVIEW functions accept a **format string** input, which controls the behavior of the function. A format string is composed of one or more format specifiers, which determine what action to take to process a given parameter. The Format Into String and Scan From String functions can use multiple format specifiers in the format string, one for each resizable input or output to the function. Characters in the string which are not part of the format specifier are copied verbatim to the output string (in the case of Format Into String) or are matched exactly in the input string (in the case of Scan From String), with the exception of special escape codes. You can use these codes to insert nondisplayable characters, the backslash, and percent characters within any format string. These codes are similar to those used in the C programming language. The following table describes the special escape codes and their meanings.

Code	Meaning
\r	Carriage Return
\t	Tab
\b	Backspace
\n	Newline
\f	Form Feed
\s	space
\xx	character with hexadecimal ASCII code xx (using 0 through 9 and upper case A through F)
\\	\
%%	%

A code does not exist for the platform-dependent end-of-line (eol) character. If you need to append one, use the End-of-Line constant from the **String** palette.

Notice also that for the Scan From String and Format & Strip functions, a space in the format string matches any amount of whitespace (spaces, tabs, and form feeds) in the input string.

The Format & Append, Format & Strip, Array To Spreadsheet String, and Spreadsheet String To Array functions use only one format specifier in the format string, because these functions have only one input that can be converted. Any extraneous specifiers inserted into these functions are treated as literal strings with no special meaning.

For functions that output a string, such as Format Into String, Format & Append, and Array To Spreadsheet String, a format specifier has the following syntax. Double brackets (*[]*) enclose optional elements.

%[-][+][^][0][Width][.Precision][{unit}]Conversion Code

For functions that scan a string, such as Scan From String, Format & Strip, and Spreadsheet String to Array, a format specifier has the following, simplified syntax.

%[Width]Conversion Code

The following table explains the elements of this syntax.

Syntax Element	Description
%	Begins the formatting specification.
- (optional)	Causes the parameter to be left justified rather than right justified within its width.
+ (optional)	For numeric parameters, includes the sign even when the number is positive.
^ (optional	When used with the e or g conversion codes, uses engineering notation (exponent is always a multiple of 3).
0 (optional)	Pads any excess space to the left of a numeric parameter with 0s rather than spaces.
Width (optional)	When scanning, specifies an exact field width to use. LabVIEW scans only the specified number of characters when processing the parameter. When formatting, specifies the minimum character field width of the output. This is not a maximum width; LabVIEW uses as many characters as necessary to format the parameter without truncating it. LabVIEW pads the field to the left or right of the parameter with spaces, depending on justification. If Width is missing or zero, the output is only as long as necessary to contain the converted input parameter.
.	Separates Width from Precision.
Precision (optional)	For floating-point parameters, specifies the number of digits to the right of the decimal point. If Width is not followed by a period, LabVIEW inserts a fractional part of six digits. If Width is followed by a period, and

	Precision is missing or 0, LabVIEW does not insert a fractional part. For string parameters, specifies the maximum width of the field. LabVIEW truncates strings longer than this length.
{unit} (optional)	Overrides the choice of unit of a VI when converting a physical quantity (a value with an associated unit). Must be a valid unit.
Conversion Codes	Single character that specifies how to convert **number**, as follows `d` to decimal integer `x` to hex integer `o` to octal integer `b` to binary integer `f` to floating-point number with fractional format `e` to floating-point number with scientific notation `g` to floating-point number using `e` format if the exponential is less than -4 or greater than Precision, for `f` format otherwise `s` to string An l (lowercase L) preceding the conversion character is ignored and u is equivalent to d. The type of the data determines whether integers are signed or unsigned. ConversionCharacter can be upper or lower case.

The Conversion Codes used in LabVIEW are similar to those used in the C programming language. However, LabVIEW uses conversion codes to determine the textual format of the parameter, not the datatype of the function.

You can use the `d`, `x`, `o`, `b`, `f`, `e` and `g` conversion codes to process any numeric LabVIEW datatype, including complex numbers and enums.

For complex numbers, you can use the format specifier to process both the real and imaginary parts.

You can use the `s` conversion code to process string or path parameters or enums.

Notice that you can use either a numeric or string conversion code with an enum, depending on whether you want the numeric value or symbolic (string) value of the enum.

For compatibility with C, LabVIEW treats a `u` conversion code (unsigned integer) the same as a `d`, and ignores an `l` or `L` preceding the conversion code. However, in LabVIEW it is the datatype of the parameter that determines the size of an integer and whether the integer is signed or unsigned.

For examples of format string usage, see the Format Into String and Scan From String functions.

Polymorphism for String Functions

String Length, To Upper Case, To Lower Case, Reverse String, and Rotate String accept strings, clusters, arrays of strings, and arrays of clusters. To Upper Case and To Lower Case also accept numbers, clusters of numbers, and arrays of numbers, interpreting them as ASCII codes for characters (refer to ASCII Codes, for the numbers that correspond to each character). Width and precision inputs must be scalar.

Carriage Return

Consists of a constant string containing the ASCII CR value.

Empty String

Consists of a constant string that is empty. Length is zero.

End of Line

Consists of a constant string containing the platform-dependent, end of line value. For Windows, the value is CRLF; for Macintosh, the value is CR; and on Unix, the value is LF.

Line Feed

Consists of a constant string containing the ASCII LF value.

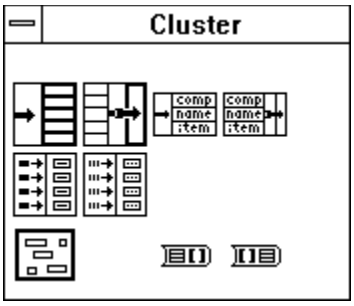
Tab

Consists of a constant string containing the ASCII HT (horizontal tab) value.

Cluster Function Descriptions

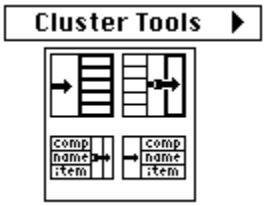
This topic describes the functions for cluster operations. Click here to access the Cluster Function Overview topic.

The following illustration shows the **Cluster** palette, which you access by selecting **Functions»Cluster**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Array to Cluster
Build Cluster Array
Bundle
Bundle By Name
Cluster To Array
Index & Bundle Cluster Array
Unbundle
Unbundle By Name

Some of the cluster functions are also available from the **Cluster Tools** palette of most terminal or wire pop-up menus, shown in the illustration that follows, if the datatype of the terminal or wire is a cluster. If you select the functions from this palette, they appear with the correct number of terminals to wire to the object on which you popped up.



For examples of cluster functions, see `examples\general\arrays`.

Array To Cluster

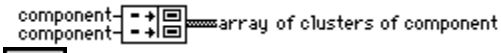
Converts a 1D array to a cluster of elements of the same type as the array elements. Pop up on the node or resize it to set the number of elements in the cluster. The default is nine. The maximum cluster size for this function is 256.

array.

cluster. Each element in **cluster** is the same as the corresponding element in **array**. The **cluster** order matches the order of the elements in the **array**.

Build Cluster Array

Assembles all the **component** inputs in top-down order into an array of clusters of that **component**. There are four, single-precision, floating-point components. The output is a four-element array of clusters containing one single-precision, floating-point number. Element 0 of the array has the value of the top component, and so on.

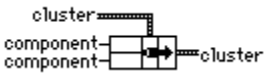


component. All of the **component** inputs must be of the same type as the value wired to the topmost **component** terminal. See the Polymorphism for Cluster Functions topic for more information.

array of clusters of component.

Bundle

Assembles all the individual input components into a single cluster or changes the values of wired components.

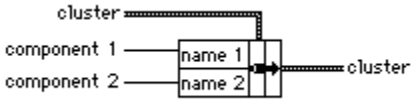


component, **component**, and **cluster**. The input parameters of this function are polymorphic--they can be of any datatype. If you wire **cluster**, only those components you want to change must be wired. The unwired components remain unchanged. If you leave **cluster** unwired, you must wire all the components.

cluster is the cluster whose value you want to change.

Bundle By Name

Replaces components in an existing cluster. After you wire the node to a cluster, you pop-up on the name terminals to choose from the list of components of the cluster.



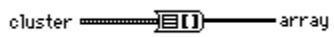
You must always wire the **cluster** input. If you are creating a cluster for a cluster indicator, you can wire a local variable of that indicator to the **cluster** input. If you are creating a cluster for a cluster control of a subVI, you can place a copy of that control (possibly hidden) on the front panel of the VI and wire the control to the **cluster** input.

component 1, **component 2**, and so on, are elements within a cluster whose labels are name 1, name 2, and so on. Select the correct name for the component from the pop-up menu on the name terminal.

cluster is the cluster whose value you want to change.

Cluster To Array

Converts a cluster of identically typed components to a 1D array of elements of the same type.

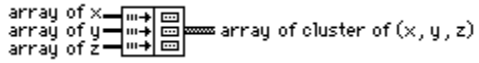


cluster. The **cluster** components cannot be arrays.

array. The elements in **array** are of the same type as the elements in **cluster**. The order of the elements in **array** are the same as the **cluste**r order of the objects.

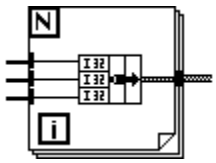
Index & Bundle Cluster Array

Indexes a set of arrays and creates a cluster array in which the *i*th element contains the *i*th element of each input array.



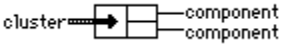
array of x, **array of y**, **array of z** can be 1D arrays of any type and do not all have to be the same type.

array of cluster of (x, y, z) is a cluster array containing one element from each input array. This function is equivalent to the following block diagram and is useful for converting a cluster of arrays to an array of clusters.



Unbundle

Disassembles a cluster into its individual components.

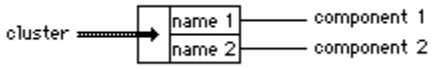


cluster is the cluster whose elements you want to access.

component is any datatype. The output parameters of this function are polymorphic. The function arranges the components from top-to-bottom in cluster order with element 0 at top. For more information on cluster order, see Setting the Order of Cluster Elements.

Unbundle By Name

Returns the cluster elements whose names you specify. You select the element you want to access by popping up on the name output terminals and selecting a name from the list of elements in the cluster.



cluster is the cluster whose elements you want to access.

component 1 and **component 2** are the elements of **cluster** you want to access by name. Select the correct name from the pop-up menu on the name terminal.

Array To Cluster Function

Array To Cluster

Build Cluster Array Function

Build Cluster Array

Bundle Function

Bundle

Bundle By Name Function

Bundle By Name

Cluster To Array Function

Cluster To Array

Index & Bundle Cluster Array Function

Index & Bundle Cluster Array

Unbundle Function

Unbundle

Unbundle By Name Function

Unbundle By Name

Cluster Constant

Use this to supply a constant cluster value to the block diagram. The elements of the cluster can be different types. You can define the cluster type by choosing any constant listed in the **Functions** palette and dragging that constant into the cluster shell. Enter values for each cluster element by using the Operating tool.

The value of the cluster constant cannot be changed while the VI executes.

You can assign a label to this constant.

Cluster Function Overview

Click here to access the Cluster Function Descriptions topic.

General Behavior of Cluster Functions
Polymorphism for Cluster Functions

General Behavior of Cluster Functions

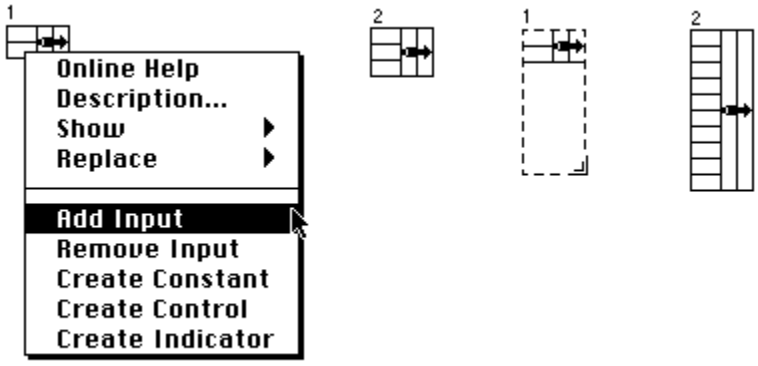
The following cluster functions are expandable.

The **Bundle By Name** and **Unbundle By Name** functions give you more flexible access to data in clusters. With these functions, you can access specific elements in clusters by name and access only the elements you want to access. Because these functions reference components by name and not by cluster position, you can change the data structure of a cluster without breaking wires, as long as you do not change the name of or remove the component you reference on the block diagram.

Some of the cluster functions have a variable number of terminals. When you drop a new function of this kind, it appears on the block diagram with only one or two terminals. You can add and remove terminals by using the pop-up menu **Add Input** or **Remove Input** options or by resizing the node using the Positioning tool. If you want to add terminals by popping up, you must place your cursor on the input terminal to access the pop-up menu.

You can shrink the node if doing so does not delete wired terminals. The **Add Input** option inserts a terminal directly after the one on which you popped up. The **Remove Input** option removes the terminal on which you popped up, even if it is wired.

The following illustration shows the two ways to add more terminals to the Bundle function.



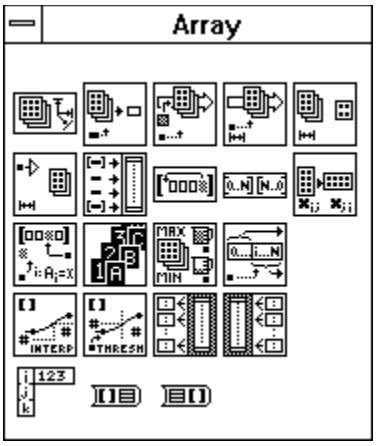
Polymorphism for Cluster Functions

The Bundle and Unbundle functions do not show the datatype for their individual input or output terminals until you wire objects to these terminals. When you wire them, these terminals look similar to the datatypes of the corresponding front panel control or indicator terminals.

Array Function Descriptions

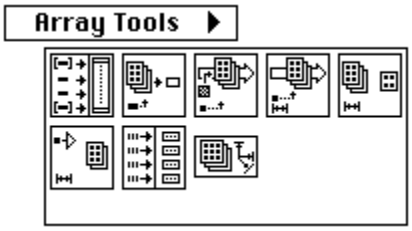
This topic describes the functions for array operations. Click here to access the Array Function Overview topic.

The following illustration shows the **Array** palette which you access by selecting **Functions»Array**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Array Max & Min
Array Size
Array Subset
Array To Cluster
Build Array
Cluster To Array
Decimate 1D Array
Index Array
Initialize Array
Interleave 1D Arrays
Interpolate 1D Array
Replace Array Element
Reshape Array
Reverse 1D Array
Rotate 1D Array
Search 1D Array
Sort 1D Array
Split 1D Array
Threshold 1D Array
Transpose 2D Array

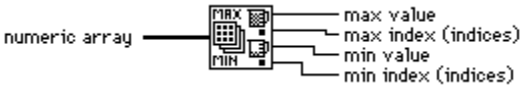
Some of the array functions are also available from the **Array Tools** palette of most terminal or wire pop-up menus, shown in the following illustration, if the datatype of the terminal or wire is an array. If you select the functions from this palette, they appear with the correct number of terminals to wire to the object on which you popped up.



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

Array Max & Min

Searches for the first maximum and minimum values in **numeric array**. This function also returns the indices where it finds the maximum and minimum values.



numeric array can have any number of dimensions. See the Polymorphism for Array Functions topic for more information.

max value and **min value** are of the same datatype as the elements in **numeric array**.

max index and **min index** are the indices for the first max and min values. If **numeric array** is multidimensional, then **max index** and **min index** are arrays, each of whose elements is the index for the corresponding dimension in **numeric array**.

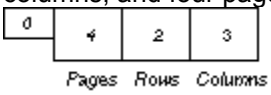
The function compares each datatype according to the rules discussed in the Rules for Comparison topic.

Array Size

Returns the number of elements in each dimension of **array**.

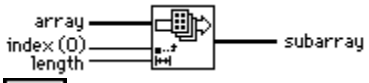
array can be an *n*-dimensional array of any type. See the Polymorphism for Array Functions topic for more information.

size(s). If **array** is one-dimensional (1D), the size is a long integer. If **array** is multidimensional, the returned value is a 1D array in which each element is a long integer representing the number of elements in the corresponding dimension of **array**. For example, a 3-D array with two rows, three columns, and four pages has the following output array.



Array Subset

Returns a portion of **array** starting at **index** and containing **length** elements.



array can be an *n*-dimensional array of any type. If **array** is multidimensional, you must add a pair of **index** and **length** terminals for each dimension using the Positioning tool or by popping up on the node and selecting **Add Dimension**. See the Polymorphism for Array Functions topic for more information.

index (0) must be a scalar number. If **index** is less than 0, the function sets it to 0. If **index** is greater than or equal to the array size, the function returns an empty array. The function coerces **index** to

a 32-bit integer if you wire another representation to it.

length must be a scalar number. If **index** plus **length** is larger than the size of the array, the function returns only as much data as is available. The function coerces **length** to a 32-bit integer if you wire another representation to it.

subarray is of the same type as the elements of **array**.

Array To Cluster

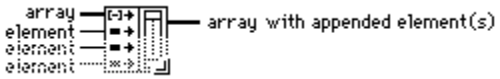
Converts a 1D array to a cluster of elements of the same type as the array elements. Pop up on the node or resize it to set the number of elements in the cluster. The default is nine. The maximum cluster size for this function is 256.

array.

cluster. Each element in **cluster** is the same as the corresponding element in **array**. The **cluster** order matches the order of the elements in the **array**.

Build Array

Appends any number of array or element inputs in top-to-bottom order to create **array with appended element**.



array can be an n-dimensional array of any type. The elements of **array** can be of any datatype. See the Polymorphism for Array Functions topic for more information.

element must be of the same datatype as the elements of array.

array with appended element(s).

Initially, this function has two element inputs, but you can adjust it to any number you want. To change an element input to an array input, pop-up on the input and select **Change to Array**. In general, to build an array of *n*-dimensions, each **array** input must be of the same dimension, *n*, and each **element** input must have *n*-1 dimensions. To create a 1D array, connect scalar values to the element inputs and 1D arrays to the array inputs. To build a 2D array, connect 1D arrays to element inputs and 2D arrays to the array inputs.

Cluster To Array

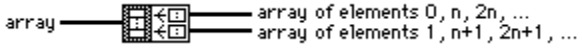
Converts a cluster of identically typed components to a 1D array of elements of the same type.

cluster. The **cluster** components cannot be arrays.

array. The elements in **array** are of the same type as the elements in **cluster**. The order of the elements in **array** are the same as the **cluste**r order of the objects.

Decimate 1D Array

Divides the elements of **array** into the output arrays, much like the way a dealer distributes cards.



array can be a 1D array of any type. If the number of elements in **array** is not an even multiple of the number of output arrays, the function does not store the extra elements in the output arrays. For example, if **array** has 18 elements, and you wire four output arrays, each array receives 4 elements, and **array** ignores elements 16 and 17.

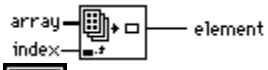
array of elements 0, n, 2n, ... and **array of elements 1, n+1, 2n+1, ...**. The function stores

array[0] at index 0 of the first output array, **array**[1] is stored at index 0 of the second output array, **array**[*n*-1] at index 0 of the last output array, **array**[*n*] at index 1 of the first output array, and so on, where *n* is the number of output terminals for this function.

For example, assume that **array** has 16 elements and that you wire four output arrays. The first output array receives elements 0, 4, 8, and 12. The second output array receives elements 1, 5, 9, and 13. The third output array receives elements 2, 6, 10, and 14. The last output array receives elements 3, 7, 11, and 15.

Index Array

Returns the **element** of **array** at **index**. If **array** is multidimensional, you must add additional **index** terminals for each dimension of the **array** by resizing the node or by popping up on the node and selecting **Add Element**.



array can be an *n*-dimensional array of any type (except array). See the Polymorphism for Array Functions topic for more information.

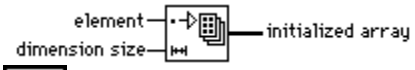
index can be a scalar number. The function coerces **index** to a 32-bit integer if you wire another representation to it. If the **index** is out of range (<0 or ³ N, where N is the size of **array**), the value of **element** is the default value of the datatype of array (zero for numbers, FALSE for Boolean controls, and empty for string).

element has the same type as the elements of **array**.

In addition to extracting an element of the array, you can *slice* out a higher dimensional component by disabling one or more of the index terminals.

Initialize Array

Creates an *n*-dimensional array in which every element is initialized to the value of **element**.



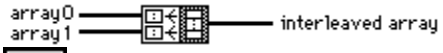
element determines the type of the array and the initial value of each array element. **element** can be any datatype except an array.

dimension size. You must add a dimension size terminal for each dimension of the initialized array. The function coerces **dimension size** to a 32-bit integer if you wire another representation to it. The function creates an empty array if any dimension size is 0.

initialized array is an array of the same type as the type you wire to **element**.

Interleave 1D Arrays

Interleaves corresponding elements from the input arrays into a single output array.



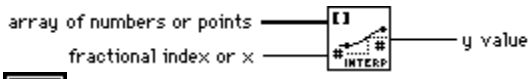
array0 and **array1** must be 1D. If the input arrays are not the same size, then the number of elements in **interleaved array** equals the number of elements in the smallest input array multiplied by the number of input arrays.

interleaved array[0] contains **array0**[0], **interleaved array**[1] contains **array1**[0], **interleaved array**[*n*-1] contains **array*n*-1**[0], **interleaved array**[*n*] contains **array0**[1], and so on, where *n* is the number of input terminals.

Interpolate 1D Array

Uses the integer part of the **fractional index or x** to index the array. Interpolate 1D Array uses the fractional part of **fractional index or x** to linearly interpolate between the values of the indexed element

and its adjacent element.



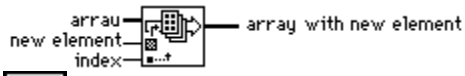
array of numbers or points can be an array of numbers or an array of points where each point is a cluster of x and y coordinates. If this input is an array of points, the function uses the first element in the cluster (x) to obtain a fractional index by linear interpolation. The function then uses this fractional index to compute the output **y value** from the second cluster element (y).

fractional index or x. If **fractional index or x** is an integer, **y value** is the value of the element at that index in **array of numbers or points**. Otherwise, the function uses the fractional part of **fractional index or x** to linearly interpolate between two array values to compute **y value**. For example, assume that **array of numbers or points** has two elements, 1 and 2. If **fractional index or x** is 0.5, **y value** is 1.5. If **fractional index or x** is less than the first x value, **y value** is the first y value. If **fractional index or x** is greater than the last x value, **y value** is the last y value.

y value.

Replace Array Element

Replaces with **new element** the element in **array** at **index**.



array can be an *n*-dimensional array of any type. See the Polymorphism for Array Functions topic for more information.

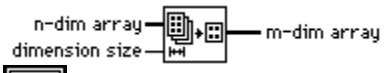
new element must be of the same type as the other elements in **array.**

index must be scalar. If **array** is multidimensional, you must wire an **index** terminal for each dimension. The function coerces **index** to a 32-bit integer if you wire another representation to it. If index is out of range (<0 or ³ N, where N is the dimension size), the function passes **array** through unchanged.

array with new element is the same type as **array**.

Reshape Array

Changes the dimension of an array according to the value of **dimension size**. For example, you can use this function to change a 1D array into a 2D array or vice versa. You can also use it to increase and decrease the size of a 1D array.



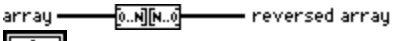
n-dim array can be an *n*-dimensional array of any type, except boolean. See the Polymorphism for Array Functions topic for more information.

dimension size. You must have *m* **dimension size** terminals for *m* dimensions. **dimension size** must be a scalar number. The function coerces **dimension size** to a 32-bit integer if you wire another representation to it.

m-dim array. If the product of the dimension sizes is greater than the number of elements in the input array, the function pads the new array with the default value of the datatype of **n-dim array**. If the product of the dimension sizes is less than the number of elements in the input array, the function truncates the array.

Reverse 1D Array

Reverses the order of the elements in **array**.

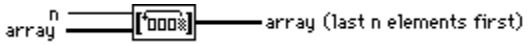


array can be a 1D array of any type. See the Polymorphism for Array Functions topic for more information.

reversed array. If **array** has *n* elements, **array**[0] becomes **reversed array**[*n*-1], **array**[1] becomes **reversed array**[*n*-2], and so on.

Rotate 1D Array

Rotates the elements of **array** the number of places and in the direction indicated by **n**.



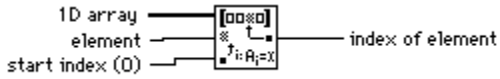
n must be a numeric datatype. The function coerces **n** to a 32-bit integer if you wire another representation to it.

array can be a 1D array of any type. See the Polymorphism for Array Functions topic for more information.

array (last n elements first). For example, if **n** is 1, the input **array**[0] becomes the output **array**[1], input **array**[1] becomes output **array**[2], and so on, and input **array**[*n*-1] becomes output **array**[0]. If **n** is -2, input **array**[0] becomes output **array**[*n*-2], input **array**[1] becomes output **array**[*n*-1], and so on, and input **array**[*n*-1] becomes output **array**[*n-3].*

Search 1D Array

Searches for **element** in **1D array** starting at **start index**.



1D array can be a 1D array of any type. See the Polymorphism for Array Functions topic for more information.

element determines the type of the array and the initial value of each array element. **element** must be scalar.

start index (0) must be a scalar number. **start index** defaults to 0 if you do not wire it.

index of element is the index where **element** is found. If the function does not find **element**, **index of element** is -1.

Sort 1D Array

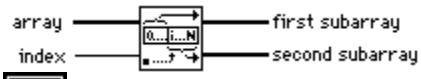
Returns a sorted version of **array** with the elements arranged in ascending order. The rules for comparing each datatype are described in the Comparison Functions topic.

array can be a 1D array of any type, except Boolean. See the Polymorphism for Array Functions for more information on what types a 1D array can be.

sorted array.

Split 1D Array

Divides **array** at **index** and returns the two portions.



array can be a 1D array of any type. See the Polymorphism for Array Functions topic at for more information.

index must be scalar. If **index** is negative or 0, **first subarray** is empty. If **index** is equal to or greater than the size of **array**, **second subarray** is empty.

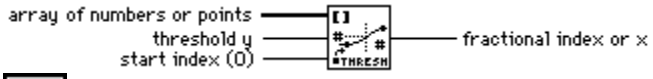
first subarray and **second subarray** contain elements of the same datatype as the elements of **array**. **first subarray** contains **array**[0] through **array**[**index**-1], and **second subarray** contains the rest

of **array**.

Threshold 1D Array

Compares **threshold y** to the values in **array of numbers or points** starting at **start index** until it finds a pair of consecutive elements such that **threshold y** is greater than the value of the first element and less than or equal to the value of the second element.

The function then calculates the fractional distance between the first value and **threshold y** and returns the fractional index at which **threshold y** would be placed within **array of numbers or points** using linear interpolation.



array of numbers or points can be a 1D array of numbers or of (x,y) points.

threshold y. If **threshold y** is less than or equal to the array value at **start index**, the function returns **start index** for **fractional index or x**. If **threshold y** is greater than every value in the array, the function returns the index of the last value. If the array is empty, the function returns NaN.

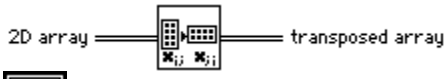
start index (0) must be scalar. If you do not wire **start index**, it defaults to 0.

fractional index or x is the interpolated result LabVIEW calculates for the **array of numbers or points** 1D input array.
For example, suppose **array of numbers or points** is an array of four numbers [4, 5, 5, 6], **start index** is 0, and **threshold y** is 5. The **fractional index or x** is 1, corresponding to the index of the first value of 5 the function finds. Suppose the array elements are 6, 5, 5, 7, 6, 6, the **start index** is 0, and the **threshold y** is 6 or less. The output is 0. If **threshold y** is greater than 7 for the same set of numbers, the output is 5. If **threshold y** is 14.2, **start index** is 5, and the values in the array starting at index 5 are 9.1, 10.3, 12.9, and 15.5, **threshold y** falls between elements 7 and 8 because 14.2 is midway between 12.9 and 15.5. The value for **fractional index or x** is 7.5, that is, halfway between 7 and 8.

If the array input consists of an array of points where each point is a cluster of x and y coordinates, the output is the interpolated x value corresponding to the interpolated position of **threshold y** rather than the fractional index of the array. If the interpolated position of **threshold y** is midway between indices 4 and 5 of the array with x values of -2.5 and 0 respectively, the output is not an index value of 4.5 as it would be for a numeric array, but rather an x value of -1.25.

Transpose 2D Array

Rearranges the elements of **2D array** such that **2D array**[*i*,*j*] becomes **transposed array**[*j*,*i*].



2D array can be a 2D array of any type. See the Polymorphism for Array Functions topic for more information.

transposed array.

Array Max & Min Function

Array Max & Min

Array Size Function

Array Size

Array Subset Function

Array Subset

Array To Cluster Function

Array To Cluster

Build Array Function

Build Array

Cluster To Array Function

Cluster To Array

Decimate 1D Array Function

Decimate 1D Array

Index Array Function

Index Array

Initialize Array Function

Initialize Array

Interleave 1D Arrays Function

Interleave 1D Arrays

Interpolate 1D Array Function

Interpolate 1D Array

Replace Array Element Function

Replace Array Element

Reshape Array Function

Reshape Array

Reverse 1D Array Function

Reverse 1D Array

Rotate 1D Array Function

Rotate 1D Array

Search 1D Array Function

Search 1D Array

Sort 1D Array Function

Sort 1D Array

Split 1D Array Function

Split 1D Array

Threshold 1D Array Function

Threshold 1D Array

Transpose 2D Array Function

Transpose 2D Array

Array Constant

Use this to supply a constant array value to the block diagram. The elements of the array must be the same type. You can define the array type by choosing any constant listed in the **Functions** palette and dragging that constant into the array shell. Enter values for each array element by using the Operating tool.

The value of the array constant cannot be changed while the VI executes.

You can assign a label to this constant.

Array Function Overview

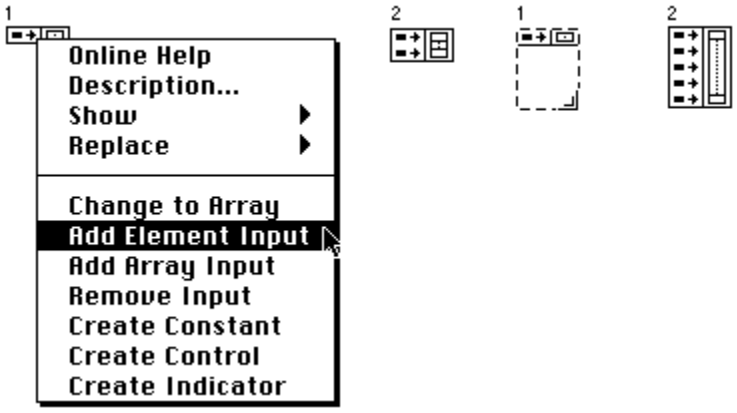
Click here to access the Array Function Descriptions topic.

General Behavior of Array Functions
Out-of-Range Index Values
Polymorphism for Array Functions

General Behavior of Array Functions

Some of the array functions have a variable number of terminals. When you drop a new function of this kind, it appears on the block diagram with only one or two terminals. You can add and remove terminals by using the pop-up menu **Add Element Input** or **Add Array Input** and **Remove Input** commands (the actual names depend on the function) or by resizing the node vertically from any corner. If you want to add terminals by popping up, you must place your cursor on the input terminals to access the pop-up menu.

You can shrink the node if doing so does not delete wired terminals. The **Add Element Input** or **Add Array Input** command inserts a terminal directly after the one on which you popped up. The **Remove Input** command removes the terminal on which you popped up, even if it is wired. The following illustration shows the two ways to add more terminals to the Build Array function.



Out-of-Range Index Values

Attempting to index an array beyond its bounds results in a default value determined by the array element type.

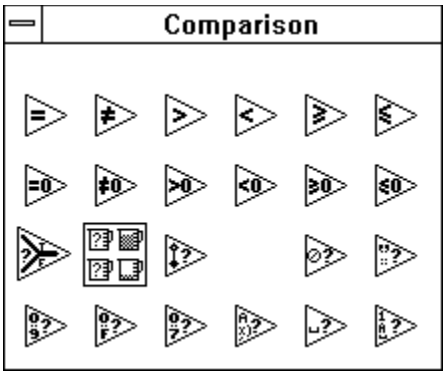
Polymorphism for Array Functions

Most of the array functions accept n-dimensional arrays of any type, although the wiring diagrams in the function descriptions show numeric arrays as the default data type.

Comparison Function Descriptions

This topic describes the functions that perform comparisons or conditional tests. Click here to access the Comparison Function Overview topic.

The following illustration shows the **Comparison** palette, which you access by selecting **Functions»Comparison**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Decimal Digit?
Empty String/Path?
Equal?
Equal To 0?
Greater?
Greater Or Equal?
Greater Or Equal To 0?
Greater Than 0?
Hex Digit?
In Range?
Less?
Less Or Equal?
Less Or Equal To 0?
Less Than 0?
Lexical Class
Max & Min
Not A Number/Path/Refnum?
Not Equal?
Not Equal To 0?
Octal Digit?
Printable?
Select
Whte Space

For examples of comparison functions, see `examples\ general\struct.llb`.

Decimal Digit?

Returns TRUE if **char** is a decimal digit ranging from 0 through 9. Otherwise, this function returns FALSE.

char can be a scalar string or number, clusters of strings or numbers, arrays of strings or

numbers, and so on. See the Polymorphism for Comparison Functions topic for more information.

digit? is a Boolean value of the same data structure as **char**.

Empty String/Path?

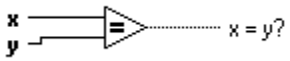
Returns TRUE if **string/path** is an empty string or path. Otherwise, this function returns FALSE.

string/path can be a scalar string or path, clusters of strings or paths, arrays of strings or paths, and so on. See the Polymorphism for Comparison Functions topic for more information.

empty? is a Boolean value of the same data structure as **string/path**.

Equal?

Returns TRUE if **x** is equal to **y**. Otherwise, this function returns FALSE. This function accepts complex numbers.



x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for more information.

x = y?. When you compare arrays, **x = y?** is a scalar when you use the **Compare Aggregates** mode and a Boolean array when you use the **Compare Elements** mode (default).

Equal To 0?

Returns TRUE if **x** is equal to 0. Otherwise, this function returns FALSE.

x can be a numeric scalar, cluster, or array of numbers. See the Polymorphism for Comparison Functions topic for more information.

x = 0? is a Boolean value of the same data structure as **x**.

Greater?

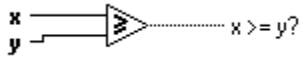
Returns TRUE if **x** is greater than **y**. Otherwise, this function returns FALSE.

x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for more information.

x > y?. When you compare arrays, **x > y?** is a scalar when you use the **Compare Aggregates** mode and a Boolean array when you use the **Compare Elements** mode (default).

Greater Or Equal?

Returns TRUE if **x** is greater than or equal to **y**. Otherwise, this function returns FALSE.



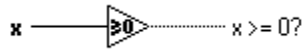
x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for more information.

x >= y?. When you compare arrays, **x >= y?** is a scalar when you use the **Compare Aggregates**

mode and a Boolean array when you use the **Compare Elements** mode (default).

Greater Or Equal To 0?

Returns TRUE if **x** is greater than or equal to 0. Otherwise, this function returns FALSE.

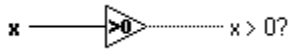


x can be a numeric scalar, cluster, or array of numbers. See the Polymorphism for Comparison Functions topic for more information.

x >= 0? is a Boolean value of the same data structure as **x**.

Greater Than 0?

Returns TRUE if **x** is greater than 0. Otherwise, this function returns FALSE.



x can be a numeric scalar, cluster, or array of numbers. Refer to the Polymorphism for Comparison Functions topic for more information.

x > 0? is a Boolean value of the same data structure as **x**.

Hex Digit?

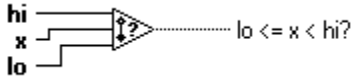
Returns TRUE if **char** is a hex digit ranging from 0 through 9, A through F, or a through f. Otherwise, this function returns FALSE.

char can be a scalar string or number, clusters of strings or numbers, arrays of strings or numbers, and so on. See the Polymorphism for Comparison Functions topic for more information.

hex? is a Boolean value of the same data structure as **char**.

In Range?

Returns TRUE if **x** is greater than or equal to **lo** and less than **hi**. Otherwise, this function returns FALSE.



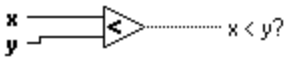
hi, **x**, and **lo** must be of the same data structure (arrays, clusters, and complex numbers), but they can have different numeric representations. For example, if you change one of the data types to an array, you must change the remaining data types to arrays to avoid bad wire connections. Refer to the Polymorphism for Comparison Functions topic for more information.

lo <= x < hi? is a Boolean scalar value.

Note: This function always operates in the Compare Aggregates mode. To produce a Boolean array as an output, you must execute this function in a loop structure.

Less?

Returns TRUE if **x** is less than **y**. Otherwise, this function returns FALSE.



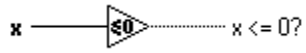
x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for

more information.

x < y?. When you compare arrays, **x < y?** is a scalar when you use the **Compare Aggregates** mode and a Boolean array when you use the **Compare Elements** mode (default).

Less Or Equal?

Returns TRUE if **x** is less than or equal to **y**. Otherwise, this function returns FALSE.



x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for more information.

x <= y?. When you compare arrays, **x <= y?** is a scalar when you use the **Compare Aggregates** mode and a Boolean array when you use the **Compare Elements** mode (default).

Less Or Equal To 0?

Returns TRUE if **x** is less than or equal to 0. Otherwise, this function returns FALSE.

x can be a numeric scalar, cluster, or array of numbers. See the Polymorphism for Comparison Functions topic for more information.

x <= 0? is a Boolean value of the same data structure as **x**.

Less Than 0?

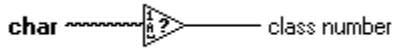
Returns TRUE if **x** is less than 0. Otherwise, this function returns FALSE.

x can be a numeric scalar, cluster, or array of numbers. See the Polymorphism for Comparison Functions topic for more information.

x < 0? is a Boolean value of the same data structure as **x**.

Lexical Class

Returns the **class number** for **char**.



char.

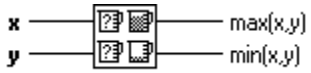
class number. The following table shows the lexical classes that correspond to the values of **class number**. See ASCII Codes for the numbers that correspond to each character.

Class Number	Lexical Class
0	Extended characters with a Command- or Option- key prefix (codes 128 through 255)
1	Nondisplayable ASCII characters (codes 0 to 31 excluding 9 through 13)
2	White space characters: Space, Tab, Carriage Return, Form Feed, Newline, and Vertical Tab (codes 32, 9, 13, 12, 10, and 11, respectively)

3	Digits 0 through 9
4	Uppercase characters A through Z
5	Lowercase characters a through z
6	All printable ASCII nonalphanumeric characters

Max & Min

Compares **x** and **y** and returns the larger value at the top output terminal and the smaller value at the bottom output terminal.



x and **y** must be of the same type, but they can have different numeric representations. See the Polymorphism for Comparison Functions topic for more information.

max(x, y) is the larger value.

min(x, y) is the smaller value.

Not A Number/Path/Refnum?

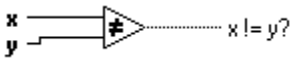
Returns TRUE if **number/path/refnum** is not a numeric value, path, or refnum. Otherwise, this function returns FALSE. NaN can be the result of dividing by 0, the square root of a negative number, and so on.

number/path/refnum can be a scalar number, path, or file refnum, or it can be a cluster or array of numbers, paths, or refnums. See the Polymorphism for Comparison Functions topic for more information.

NaN/Path/Refnum? is a Boolean value of the same data structure as **number/path/refnum**.

Not Equal?

Returns TRUE if **x** is not equal to **y**. Otherwise, this function returns FALSE. This function accepts complex numbers.

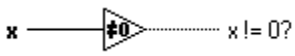


x and **y** must be of the same type. See the Polymorphism for Comparison Functions topic for more information.

x!= y?. When you compare arrays, **x!= y?** is a scalar when you use the **Compare Aggregates** mode and a Boolean array when you use the **Compare Elements** mode (default).

Not Equal To 0?

Returns TRUE if **x** is not equal to 0. Otherwise, this function returns FALSE.



x can be a numeric scalar, cluster, or array of numbers. See the Polymorphism for Comparison Functions topic for more information.

x!= 0? is a Boolean value of the same data structure as **x**.

Octal Digit?

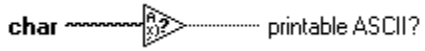
Returns TRUE if **char** is an octal digit ranging from 0 through 7. Otherwise, this function returns FALSE.

char can be a scalar string or number, clusters of strings or numbers, arrays of strings or numbers, and so on. See the Polymorphism for Comparison Functions topic for more information.

octal? is a Boolean value of the same data structure as **char**.

Printable?

Returns TRUE if **char** is a printable ASCII character. Otherwise, this function returns FALSE.



char can be a scalar string or number, clusters of strings or numbers, arrays of strings or numbers, and so on. See the Polymorphism for Comparison Functions topic for more information.

printable ASCII? is a Boolean value of the same data structure as **char**.

Select

Returns the value connected to the **t** input or **f** input, depending on the value of s. If s is TRUE, this function returns the value connected to **t**. If s is FALSE, this function returns the value connected to **f**.

Note: If t and f are arrays, use a Case structure to perform this operation in order to save memory.

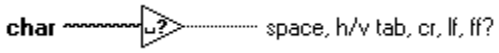
t and **f** must be of the same type, but they can have different numeric representations. See the Polymorphism for Comparison Functions topic for more information.

s determines whether the function returns the value of **t** or **f** in s**? t:f**.

s? t:f is the value wired to **t** if s is TRUE; is the value wired to **f** if s is FALSE.

White Space?

Returns TRUE if **char** is a white space character, such as space, Tab, Newline, Carriage Return, Form Feed, or Vertical Tab. Otherwise, the function returns FALSE.



char can be a scalar string or number, clusters of strings or numbers, arrays of strings or numbers, and so on. See the Polymorphism for Comparison Functions topic for more information.

space, h/v tab, cr, lf, ff? is a Boolean value of the same data structure as **char**.

Comparison Function Overview

Click here to access the Comparison Function Descriptions topic.

Rules for Comparison
Character Comparison
Polymorphism for Comparison Functions

Rules for Comparison

Most of the comparison functions test one input or compare two inputs and return a Boolean value. The functions convert numbers to the same representation before comparing them. Comparisons with a value of NaN (not a number) return a value that indicates inequality. The Numeric Conversion topic contains more information about conversion rules.

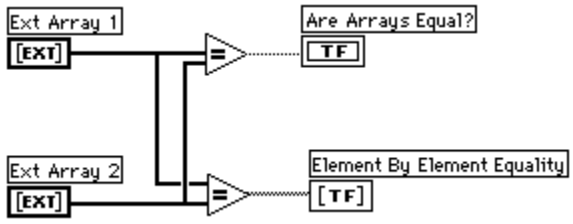
The Boolean value TRUE is greater than the Boolean value FALSE.

These functions compare strings according to the numerical equivalent of the ASCII characters. Thus, `a` (with a decimal value of 97) is greater than `A` (65), which is greater than the numeral `0`Ê(`48`), which is greater than the space character (`32`). These functions compare characters one by one from the beginning of the string until an inequality occurs, at which time the comparison ends. For example, LabVIEW evaluates the strings `abcd` and `abef` until it finds `c`, which is greater than the value of `e`. The presence of a character is greater than the absence of one. Thus, the string `abcd` is greater than `abc` because the first string is longer.

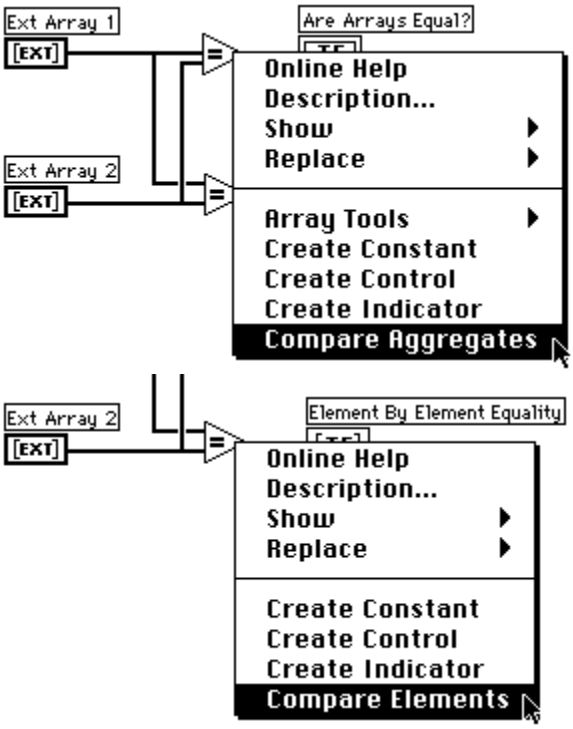
The functions that test the category of a string character (for example, the Decimal Digit? and Printable? functions) evaluate only the first character of the string.

The comparison functions compare clusters the same way they compare strings, one element at a time starting with the 0th element until an inequality occurs. Clusters must have the same number of elements, of the same type, and in the same order if you want to compare them.

Some of the comparison functions have two modes for comparing arrays or clusters. In the **Compare Aggregates** mode, if you compare two arrays or clusters, the function returns a single value. In the **Compare Elements** mode, the function compares the elements individually and then returns an array or cluster of Boolean values. The following illustration shows the two modes.



You change the comparison mode by selecting **Compare Elements** or **Compare Aggregates** in the pop-up menu for the node, as shown in the following illustrations.



When you compare two arrays of unequal lengths in the **Compare Elements** mode, LabVIEW ignores each element in the larger array whose index is greater than the index of the last element in the smaller array.

When you use the **Compare Aggregates** mode to compare two arrays, the following occurs. (1) LabVIEW searches for the first set of corresponding elements in the two inputs that differ, and uses those to determine the results of the comparison. (2) If all elements are identical except that one has more elements, LabVIEW considers the longer array to be greater than the shorter array. (3) If no elements of the two arrays differ, and the arrays have the same length, the arrays are equal. Thus, LabVIEW considers the array [1,2,3] to be greater than the array [1,2] and returns a single Boolean value in the **Compare Aggregates** mode.

When comparing clusters using the **Compare Aggregates** mode, LabVIEW goes by cluster order instead of array order. The two clusters being compared are always the same length.

In the **Compare Elements** mode, LabVIEW returns a Boolean for each of the first two elements and ignores the last element of the larger array, as in the preceding example.

Arrays must have the same dimension size (for example, both two-dimensional), and for the comparison between multidimensional arrays to make sense, each dimension must have the same size.

The comparison functions that do not have the **Compare Aggregates** or **Compare Elements** modes compare arrays in the same manner as stringsÑone element at a time starting with the 0th element until an inequality occurs.

Character Comparison

You can use the functions that compare characters to determine a character's type. The following functions are character comparison functions.

- Decimal Digit?
- Hex Digit?

- Lexical Class
- Octal Digit?
- Printable?
- White Space?

If the input is a string, the functions test the first character. If the input is an empty string, the result is FALSE. If the input is a number, the functions interpret it as a code for an ASCII character.

See ASCII Codes, for the numbers that correspond to each character.

Polymorphism for Comparison Functions

The functions Equal?, Not Equal?, and Select take inputs of any type, as long as the inputs are the same type.

The functions Greater or Equal?, Less or Equal?, Less?, Greater?, Max & Min, and In Range? take inputs of any type except complex, path, or refnum, as long as the inputs are the same type. You can compare numbers, strings, Booleans, arrays of strings, clusters of numbers, clusters of strings, and so on. You cannot, however, compare a number to a string or a string to a Boolean, and so on.

The functions that compare values to zero accept numeric scalars, clusters, and arrays of numbers. These functions output Boolean values in the same data structure as the input.

The Not A Number/Path/Refnum function accepts the same input types as functions that compare values to zero. This function also accepts paths and refnums. Not A Number/Path/Refnum outputs Boolean values in corresponding structures.

The functions Decimal Digit?, Hex Digit?, Octal Digit?, Printable?, and White Space? accept a scalar string or number input, clusters of strings or non-complex numbers, arrays of strings or non-complex numbers, and so on. The output consists of Boolean values in the same data structure as the input.

The function Empty String/Path? accepts a path, a scalar string, clusters of strings, arrays of strings, and so on. The output consists of Boolean values in the same data structure as the input.

You can use the Equal?, Not Equal?, Not A Number/Path/Refnum?, Empty String/Path?, and Select functions with paths and refnums, but no other comparison functions accept paths or refnums as inputs.

Comparison functions that use arrays and clusters normally produce Boolean arrays and clusters of the same structure. You can pop-up and change to compare aggregates, in which case the function outputs a single Boolean value. The function compares aggregates by comparing the first set of elements to produce the output, unless the first elements are equal, in which case the function compares the second set of elements, and so on.

Decimal Digit? Function

Decimal Digit?

Empty String/Path? Function

Empty String/Path?

Equal? Function

Equal?

Equal To 0? Function

Equal To 0?

Greater? Function

Greater?

Greater Or Equal? Function

Greater Or Equal?

Greater Or Equal To 0? Function

Greater Or Equal To 0?

Greater Than 0? Function

Greater Than 0?

Hex Digit? Function

Hex Digit?

In Range? Function

In Range?

Less? Function

Less?

Less Or Equal? Function

Less Or Equal?

Less Or Equal To 0? Function

Less Or Equal To 0?

Less Than 0? Function

Less Than 0?

Lexical Class Function

Lexical Class

Max & Min Function

Max & Min

Not A Number/Path/Refnum? Function

Not A Number/Path/Refnum?

Not Equal? Function

Not Equal?

Not Equal To 0? Function

Not Equal To 0?

Octal Digit? Function

Octal Digit?

Printable? Function

Printable?

Select Function

Select

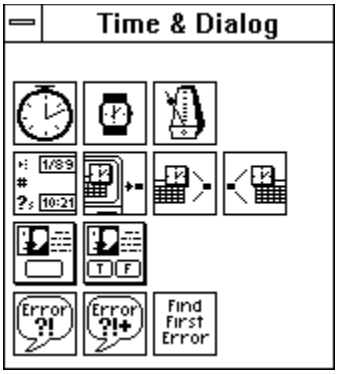
White Space Function

Whte Space

Time and Dialog Function Descriptions

This topic describes the timing functions, which you can use to get the current time, measure elapsed time, or suspend an operation for a specific period of time. Error Handling is covered in this topic as well. For general information about Time functions, see the Timing Functions topic.

The following illustration shows the **Time & Dialog** palette, which you access by selecting **Functions»Time & Dialog**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



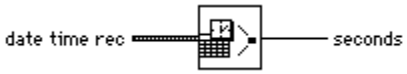
Date/Time To Seconds
Get Date/Time In Seconds
Get Date/Time String
One Button Dialog Box
Seconds To Date/Time
Tick Count (ms)
Two Button Dialog Box
Wait (ms)
Wait Till Next ms Multiple

Click here to access the Error Handling VI Descriptions.

For examples of time and dialog functions, see `viopts.llb`, located in `examples\general`.

Date/Time To Seconds

Converts a cluster of nine, signed 32-bit integers assumed to specify the local time (second, minute, hour, day, month, year, day of the week, day of the year, and Standard or Daylight Savings Time) in the configured time zone for your computer into a time-zone-independent number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.



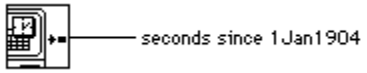
date time rec.

seconds.

If the year and month integers are out of range, the results are unpredictable. LabVIEW ignores the day of the week and day of the year integers. The other five integers can be any value. Thus, you can specify Julian dates by setting the month to January and the current day to the day of the year. For example, use January 150 for the 150th day of the year.

Get Date/Time In Seconds

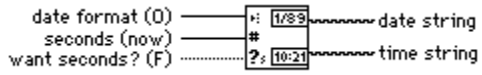
Returns a time-zone-independent number that contains the number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.



seconds since 1 Jan 1904.

Get Date/Time String

Converts a time-zone-independent number assumed to be the number of **seconds** that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time, to a date and time string in the configured time zone for your computer.



date format (0) selects the appearance of the **date string**. The following date formats are available for systems configured to use the standard, U.S. data format.

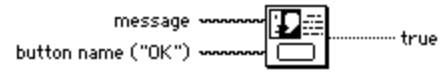
0: 1/21/94
1: Friday, January 21, 1994
2: Fri, Jan 21, 1994

These formats vary with your system configuration. Thus, your system is configured to use a data format other than the standard, U.S. format.

seconds (now). If you leave **seconds** unwired, LabVIEW uses the current date and time.
want seconds? (F) controls the display of seconds in the **time string**.
date string.
time string.

One Button Dialog Box

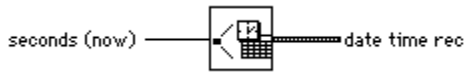
Displays a dialog box that contains a **message** and a single button. The **button name** is the name displayed on the dialog box button.



message.
button name (OK). If you leave this input unwired, **button name** defaults to OK.
true. When you click on the button, **true** contains a value of TRUE.

Seconds To Date/Time

Converts a time-zone-independent number assumed to be the number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time, to a cluster of nine, signed 32-bit integers that specify (second, minute, hour, day of the month, number of month (1-12), year, day of the week, day of the year, and Standard or Daylight Savings Time) in the configured time zone for your computer.



seconds (now). If you leave **seconds** unwired, LabVIEW uses the current date and time.
date time rec.

Tick Count (ms)

Returns the value of the millisecond timer. The base reference time (millisecond zero) is undefined; that is, you cannot convert **millisecond timer value** to a real-world time or date. Be careful when you use this function in comparisons, because the value of the millisecond timer wraps from 2^32-1 to 0.

millisecond timer value returns the value of the millisecond timer when the wait expires.

Two Button Dialog Box

Displays a dialog box that contains a **message** and two buttons. **T button name** and **F button name** are the names displayed on the buttons of the dialog box.

message.

T button name (OK). If you leave this input unwired, the name defaults to OK.

F button name (ÒCancelÓ). If you leave this input unwired, the name defaults to Cancel.

T button?. If you click on the dialog box button named **T button name**, **T button?** returns a value of TRUE. If you click on the dialog box button named **F button name**, **T button?** returns a value of FALSE.

Wait (ms)

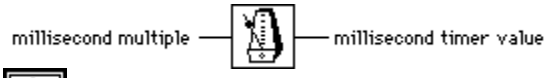
Waits the specified number of milliseconds and then returns the value.

milliseconds to wait.

millisecond timer value returns the value of the millisecond timer when the wait expires.

Wait Till Next ms Multiple

Waits until the value of the millisecond timer becomes a multiple of the specified **millisecond multiple**. You can use this function to synchronize activities. You can call this function in a loop to control the loop execution rate. However, it is possible that the first loop period may be short.



millisecond multiple.

millisecond timer value returns the value of the millisecond timer when the wait expires.

Timing Functions

Click here to access the Time and Dialog Function Descriptions topic.

The Date/Time To Seconds and the Seconds To Date/Time functions have a parameter called **date time rec,** which is a cluster that consists of the following nine signed 32-bit integers in the following order.

		Value and Range
0.	(second)	0 to 59
1.	(minute)	0 to 59
2.	(hour)	0 to 23
3.	(day of month)	1 to 31 as output from the function;1 to 366 as input

4. (month) 1 to 12
5. (year) 1904 to 2040
6. (day of week) 1 to 7 (Sunday to Saturday)
7. (day of year) 1 to 366
8. (DST) 0 to 1 (0 for Standard Time, 1 for Daylight Savings Time)

The Wait (ms) and Wait Till Next ms Multiple functions make asynchronous system calls, but the nodes themselves function synchronously. That is, they do not complete execution until the specified time has elapsed. The functions use asynchronous calls so that other nodes can execute while the timing nodes wait.

Note: National Instruments can only guarantee correct time values across all platforms for the range 2082844800 to 4230328447 seconds or 12:00 a.m., Jan. 1, 1970, Universal Time to 3:14 a.m., Jan. 19, 2038, Universal Time.

Error Handling in LabVIEW

Click here to access the Error Handling VI Descriptions topic.

For general information about Error Handling, see the Error I/O and the Error State Cluster topic.

The following are error handling VIs:

Find First Error
General Error Handler
Simple Error Handler

Every time you design a program, you should consider the possibility that something can go wrong and, if it does, you should consider how your program should manage the problem. LabVIEW automatically notifies you with a dialog box when only a few run-time errors occur, mostly for file dialog operations. It does not report all errors. If it was to report all errors, you would lose the flexibility to determine what to do when an error occurs and how and when to inform the user of the error in your program.

Rigorous error checking, especially for I/O operations (file, serial, GPIB, data acquisition, and communication), is invaluable in all phases of a project. This section describes three I/O situations in which errors can occur.

The first error can occur when you have initialized your communications incorrectly or have written improper data to your external device. This type of problem usually occurs during program development and disappears once you finish debugging your program. However, you can spend a lot of time tracking down a simple programming mistake because you have not incorporated error checks. Without error checks, all you know is that you program does not work. You do not know why the error occurred or where it is.

The second type of error can occur because your external device may be powered off, broken down, or otherwise unable to do what it normally does. This type of problem can occur at any time, but if you have incorporated error checking, LabVIEW notifies you immediately when such operational failures occur.

The third kind of error can occur when you upgrade LabVIEW or your operating system software, and you notice a bug in either a LabVIEW program or a system program. This type of error means you should check errors that you may feel safe ignoring, such as those from functions that close files or clear DAQ operations. The bottom line is, check all I/O operations for errors.

It is easy to ignore error checking when you have to add error handling code to test and report errors. The VIs described here are designed to make it easier for you to create programs with error checking and handling.

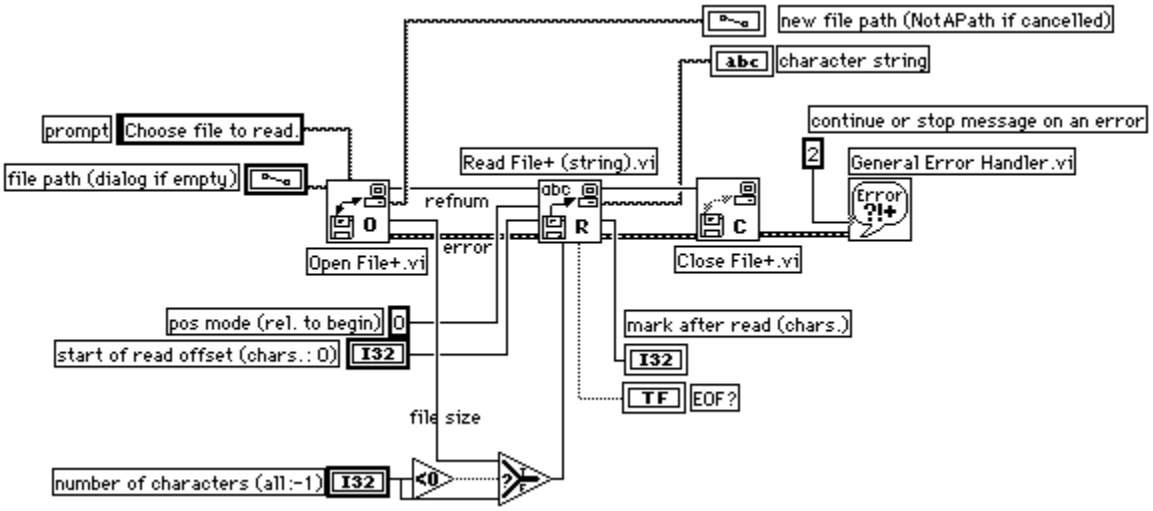
LabVIEW functions and library VIs return errors in one of two wayswith numeric error codes and with an error state cluster. Typically, functions output error codes while VIs incorporate the error cluster, usually within a framework called error input/output or error I/O.

Error I/O and the Error State Cluster

The concept of error I/O is natural to the LabVIEW dataflow architecture. If data information can flow from one node to another, so can error state information. Each node that needs to know about errors tests the incoming error state and responds appropriately. If no error exists, the node executes normally. If an error does exist, the node detects an error, skips execution, and then passes its error state out to the next node, which responds in the same way. In this fashion, notice of the first error that occurs in a sequence of operations is passed through all the nodes, with each node responding to the error. At the end of the flow, LabVIEW reports the error to the user.

Error I/O has an additional benefityou can use it to control the execution order of independent operations. While you can use the DAQ taskID to control the order of DAQ operations for one group, you cannot use it to control the order for multiple groups. The DAQ taskID does not work with other types of I/O operations such as file operations.

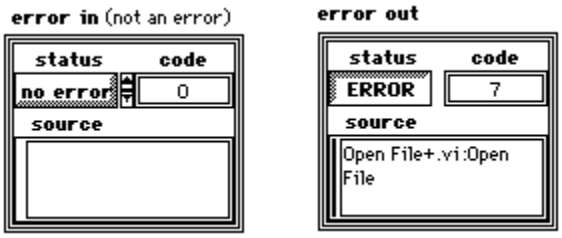
The following diagram from the File Utility VI, Read Characters From File, shows how error I/O is implemented in a simple VI.



The operation starts at `Open File+.vi`. If it opens the file successfully, `Read File+ (string).vi` reads the file and `Close File+.vi` closes the file. If you pass in an invalid path, `Open File+.vi` detects the error and passes the error state with the other two VIs to the General Error Handler, which reports it. Notice that the only presence of error handling on this block diagram is the error wire and the General Error Handler. It is neither cumbersome nor distractive.

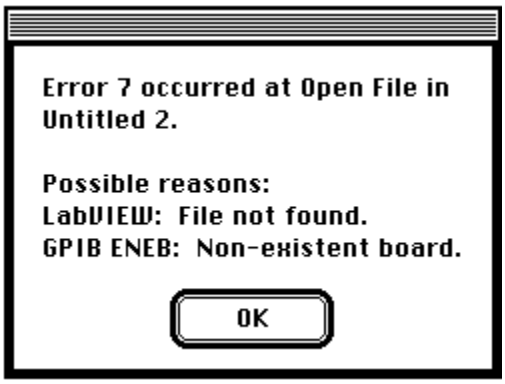
The error state consists of three pieces of information, which are combined into the error cluster. The status is a Boolean valueTRUE if an error exists, FALSE if it does not. The code consists of an unsigned 32-bit integer that identifies the error. In some cases, a non-zero error code coupled with a FALSE error status signals a warning rather than a fatal error. For example, a DAQ timeout event (code 10800) is typically reported as a warning. The source consists of a string that identifies where the error occurred.

The error in and error out state clusters for the Open File+ VI, where the error shown in the preceding example originated, are shown in the following illustration. The **error in** cluster, whose default value is *no error* does not need to be wired if it is the first in the chain.



You can find the **error in** and **error out** clusters by selecting **Controls»Array & Cluster** on the front panel.

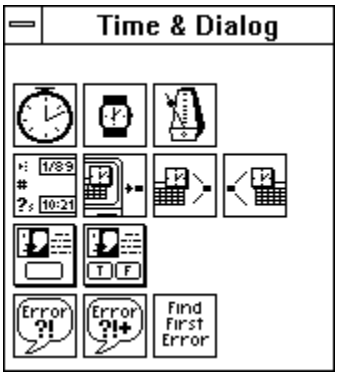
The following illustration shows the message you receive from the General Error Handler if you pass in an invalid path.



General Error Handler is one of the three error handling utility VIs. It contains a database of error codes and descriptions, from which it creates messages like the above. The Simple Error Handler performs the same basic operation but has fewer options. The third VI, Find First Error, creates the error I/O cluster from functions or VIs that output only scalar error codes.

Error Handling VI Descriptions

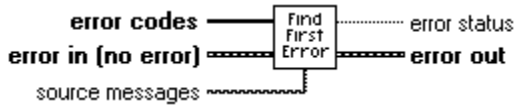
Click here to access the Error Handling in LabVIEW topic.



Find First Error
General Error Handler
Simple Error Handler

Find First Error

Tests the error status of one or more low-level functions or subVIs that output a numeric error code.



error codes is an array of the numeric error codes assembled from local subVIs or functions. If there is no error indicated in the **error in** cluster, the VI tests these codes in ascending order for non-zero values. If the VI finds a non-zero value, **error out** reflects the error status of that input.

error in (no error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the error in cluster in error out. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

source messages contains the source message you want to appear in the **error out** cluster if the VI finds an error in **error codes**. Place each message on a separate line of a string constant or control. Use of this input is optional.

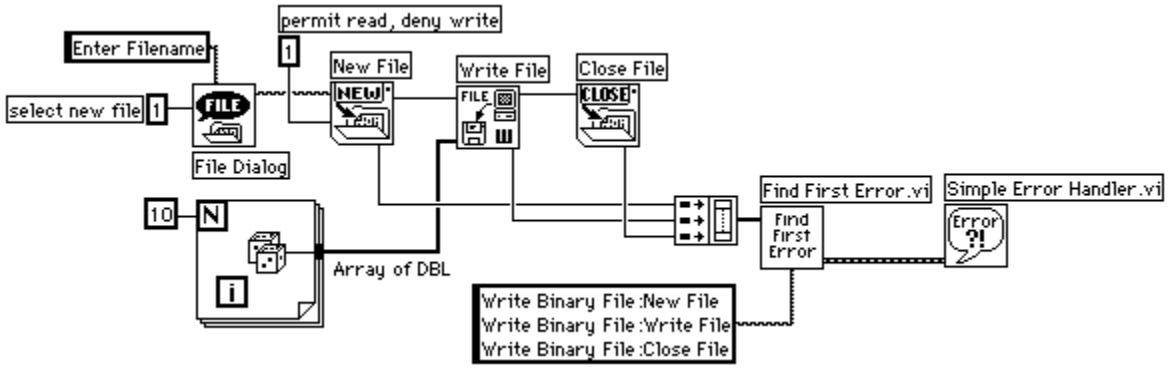
If an error occurs at Read File, the VI picks the second line for the source message of the error out cluster.

error out contains error information. If the error in cluster indicated an error, the error out cluster contains the same information. Otherwise, error out describes the error status of this VI.

error?.

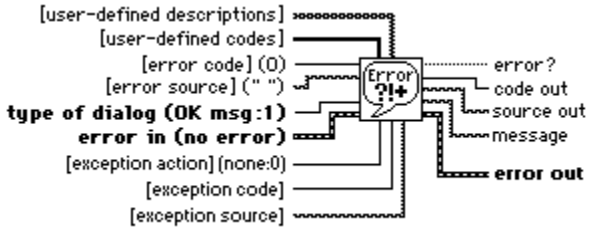
If this VI finds an error, it sets the parameters in the **error out** cluster. You can wire this cluster to the Simple or General Error Handler to identify the error and describe it to the user.

The following illustration shows how you can use Find First Error in the example VI Write Binary File. Find First Error creates the error cluster from individual error numbers, and Simple Error Handler reports any errors to the user.



General Error Handler

Determines whether an error has occurred. If an error occurred, this VI creates a description of the error and optionally displays a dialog box.



type of dialog (OK msg:1) determines what type of dialog box should be displayed, if any. Regardless of its value, the VI outputs error information and a **message** describing the error.

- 0: displays no dialog box. This is useful if you want to have programmatic control over handling the error.
- 1: (the default value) displays a dialog box with a single OK button. After the user responds, the VI returns control to the main VI.
- 2: displays a dialog box with buttons allowing the user to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.

error in (no error) describes an error that you want to check. If unwired, this VI checks **error code** for errors.

status is TRUE if an error occurred. If this value is FALSE, then the error handler assumes that no error occurred according to the **error in** and then checks the **error code**.

code is the error code number identifying the error.

source identifies where an error occurred. In case of an error, most VIs that use the error in and error out clusters set **source** to the name of the VI or function that produced the error.

[user-defined descriptions] is an array of descriptions of user-defined codes. If an incoming error matches one in **user-defined codes**, the VI uses the corresponding description from **user-defined descriptions** in **message**.

[user-defined codes] is an array of the numeric error codes you define in your VIs. The VI searches this array after searching the internal list of error codes. Codes in the range of 5000 to 9999 are reserved for users.

[error code] (0) is a numeric error code. If **error in** indicates an error, the VI ignores **error code**. If not, the VI tests it. A non-zero value signifies an error.

[error source] () is an optional string you can use to describe the source of **error code**.

[exception action] (none:0) is a way for you to create exceptions to error handling. You can treat what is normally an error as no error, or treat a no error condition as an error using this parameter.

- 0: (the default value) performs no error exception handling.
- 1: cancels an error under the following conditions:

If the VI detects an error, as described in the **status** and **error code** parameters, and if that error code value matches **exception code** and the error source value matches **exception source**, the VI sets **status out** to FALSE, **code out** to 0, and **source out** to an empty string. An empty **exception source** string matches any **error source** string.

2: sets an error under the following conditions:
If the VI detects no error, as described in the **status** and **error code** parameters, but the **code** value of **error in** matches **exception code** and the error source value matches **exception source**, the VI sets **status out** to TRUE, **code out** to the **code** value from **error in**, and **source out** to the **source** value from **error in**.

[exception code] is the error code that you want to treat as an exception. By default, it is 0.

[exception source] is the error message that you want to use to test for an exception. By default, it is an empty string.

status out is TRUE if an error was indicated.

error out contains the same information as **status out**, **code out**, and **source out**. It has the same structure as **error in**.

code out is the error code indicated by the **error in** or **error code**.

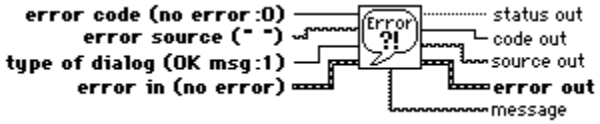
source out indicates the source of the error.

message describes the error code that occurred, the source of the error, and a description of the error.

error?.

Simple Error Handler

Determines whether an error occurred. If it finds an error, this VI creates a description of the error and optionally displays a dialog box.



error code (no error:0) is a numeric error code. The VI ignores this value if **error in** indicates an error. Otherwise it tests the value. A non-zero value signifies an error.

type of dialog (OK msg:1) determines what type of dialog box to display, if any. Regardless of its value, the VI outputs the error information and **message** describing the error.

0: displays no dialog box. This is useful if you want to have programmatic control over handling errors.

1: (the default value) displays a dialog box with a single OK button. After the user acknowledges the dialog box, the VI returns control to the main VI.

2: displays a dialog box with buttons, which the user can use to either continue or stop. If the user selects Stop, the VI calls the Stop function to halt execution.

error in (no error) describes an error that you want to check. If you leave **error in** unwired, this VI checks **error code** for errors.

status is TRUE if an error occurred. If this value is FALSE, the VI assumes that no error occurred according to the **error in**, and then checks **error code**.

code is the error code associated with an error.

source. In case of an error, most VIs that use the error in and error out clusters set **source** to the name of the VI or function that produced the error.

error source (- -**)** is an optional string that you can use to describe the source of **error code**. If **error code** indicates an error, the VI uses this string in the **message** that this VI returns and possibly displays.

 **error out** contains the same information as **status out**, **code out**, and **source out**. It has the same structure as **error in**.

 **status out** is TRUE if the VI found an error.

 **code out** is the error code indicated by **error in** or **error code**.

 **source out** indicates the source of the error.

 **message** describes the error that occurred and the source of the error.

Simple Error Handler calls General Error Handler and has the same basic functionality as General Error Handler, but with fewer options.

Date/Time To Seconds Function

Date/Time To Seconds

Get Date/Time In Seconds Function

Get Date/Time In Seconds

Get Date/Time String Function

Get Date/Time String

One Button Dialog Box Function

One Button Dialog Box

Seconds To Date/Time Function

Seconds To Date/Time

Tick Count (ms) Function

Tick Count (ms)

Two Button Dialog Box Function

Two Button Dialog Box

Wait (ms) Function

Wait (ms)

Wait Till Next ms Multiple Function

Wait Till Next ms Multiple

Simple Error Handler.vi

Simple Error Handler

General Error Handler.vi

General Error Handler

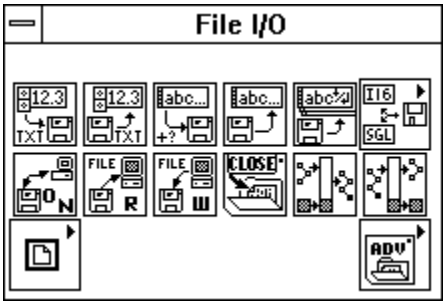
Find First Error.vi

Find First Error

File I/O Function Descriptions

This topic describes the low-level functions that manipulate files and directories. This topic also describes file constants and the high-level file VIs. Click here to access the File I/O Function Overview topic.

You access these functions, constants, and VIs by selecting **Functions»File I/O**. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Build Path
Close File
Open/Create/Replace File
Read Characters From File
Read File
Read From Spreadsheet File
Read Lines From File
Strip Path
Write Characters To File
Write File
Write To Spreadsheet File

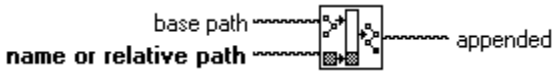
Subpalettes

Advanced File Functions
Binary File VIs
File Constants

For examples of file functions and VIs, see `examples\file`.

Build Path

Creates a new path by appending a name (or relative path) to an existing path.



name or relative path is the new path component appended to the specified path. If **name or relative path** is an empty string or an absolute or invalid path, this function sets **appended** to Not A Path.

base path specifies the path to which this function appends **name**. If you leave path unwired, it defaults to **empty path**. If **base path** is invalid, this function sets **appended** to Not A Path.

appended is the resulting path.

Close File

Writes all buffers of the file identified by **refnum** to disk, updates the directory entry of the file, closes the file, and voids **refnum** for subsequent file operations.

refnum is the file refnum associated with the file you want to close.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes regardless of whether an incoming error exists.

status is TRUE if an error occurred. If **status** is TRUE, this VI still closes the file.

code is the error code number identifying the error.

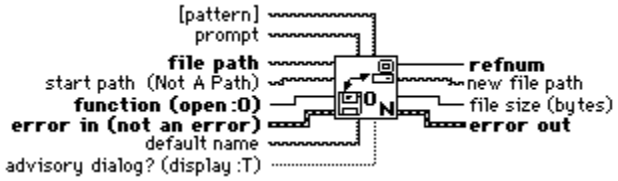
source identifies where the error occurred.

path.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Open/Create/Replace File

Opens an existing file, creates a new file, or replaces an existing file, programmatically or interactively using a file dialog box. You can optionally specify a dialog **prompt**, default file name, **start path**, or filter **pattern**. Use this VI with the intermediate Write File or Read File functions.



file path is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

function (open 0) is the operation to perform:

0: opens an existing file. Error 7 occurs if the file cannot be found (default value).
1: opens an existing file or creates a new file if one does not exist.
2: creates a new file or replaces a file if it exists and you give permission. This VI replaces a file by opening the file and setting its end of file to 0. Error 43 occurs if you do not select the replacement in an advisory dialog box.
3: creates a new file. Error 8 occurs if the file already exists.

error in (not an error) describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

[pattern] is the pattern matching specification to display only certain types of files or directories (or folders). See the description of the File Dialog function for more information.

prompt is the message that appears below the list of files and directories (or folder) in the file dialog box.

start path (Not A Path) is the path name to the initially displayed directory (or folder) in a file dialog box. The default value is Not A Path, which indicates that you should use the path to the last directory (or folder) shown in a file dialog box.

default name is the initial file name that appears in the selection box of the file dialog box.

advisory dialog? (display:T) Set to TRUE (default) if you want a dialog if function=0 and the file does not exist, or if function=2 or 3 and the file exists.

refnum is the reference number of the open file. The value is Not A Refnum if the file cannot be opened.

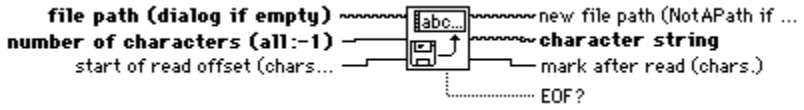
error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

new file path is the path of the file opened or created. You can use this output to determine the path of a file that you open or create using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

file size (bytes) is the size of the file in bytes; it is also the location of the end of file.

Read Characters From File

Reads a specified number of characters from a byte stream file beginning at a specified character offset. The VI opens the file before reading from it and closes it afterwards.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

number of characters (all:-1) is the maximum number of characters the VI reads. The VI reads fewer characters if it reaches the end of file first. If **number of characters** is <0, the VI reads the entire file. The default value is -1.

start of read offset (chars...) is the position in the file, measured in characters (or bytes), at which the VI begins reading.

character string is the data read from the file.

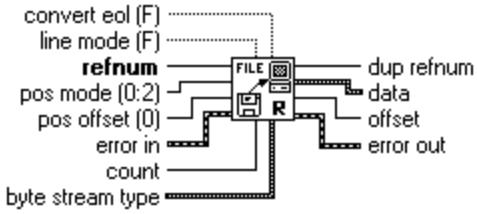
new file path (Not A Path if...) is the path of the file from which the VI reads data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

mark after read (chars.) is the location of the file mark after the read; it points to the character (byte) in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of file.

Read File

Reads data from the file specified by **refnum** and returns it in **data**. Reading begins at a location specified by **pos mode** and **pos offset** and depends on the format of the specified file.



See the topics on Reading Byte Stream Files , Reading Datalog Files , Wiring byte stream type , and When You Do Not Wire Byte Stream Type for more information on the functionality of **convert eol**, **line mode**, **count**, **byte stream type**, and **data**.

refnum is the file refnum associated with the file from which you want to read data.

pos mode (0:2), together with **pos offset**, specifies where the read operation should begin reading.

0: **pos offset** indicates that the read operation should begin at the beginning of the file plus **pos offset**.
1: **pos offset** indicates that the read operation should begin at the end of the file plus **pos offset**.
2: **pos offset** indicates that the read operation should begin at the current location of the file mark plus **pos offset**.

If the computed location does not exist in the file, for example, **pos mod** = 0 and **pos offset** = -10, the file mark does not move, no data is read, and the function returns an error.

If you wire **pos offset**, **pos mode** defaults to 0, and the offset is relative to the beginning of the file. If you do not wire **pos offset** it defaults to 0, **pos mode** defaults to 2, and the read operation starts at the current file mark.

 **pos offset (0)** specifies how far from the location specified by **pos mode** to start reading. If the file specified by **refnum** is a datalog file, you express **pos offset** in terms of records of the datatype stored in the datalog file; otherwise, you express **pos offset** in units of bytes. **pos offset** defaults to 0 if unwired.

 **error in** describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **ERROR IN** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

 **status** is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

 **code** is the error code number identifying the error.

 **source** identifies where the error occurred.

 **dup refnum** is a flow-through parameter with the same value as **refnum**.

 **offset** indicates the new location of the current file mark relative to the beginning of the file. **offset** is expressed in the same terms as **pos offset** (records for datalog files and bytes for byte stream files).

 **error out** contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Reading Byte Stream Files

If **refnum** is a byte stream file refnum, the Read File function reads data from the byte stream file specified by **refnum**. You can wire either **line mode** or **byte stream type** when you read byte stream files, but you cannot wire both. If you do not wire **byte stream type**, Read File assumes the data that begins at the designated byte offset is a string of characters. If you wire **byte stream type**, the function interprets **data** starting at the designated byte offset to be **count** instances of that type. Following the read operation, the function sets the file mark to the byte following the last byte read. If the function encounters end of file before reading all of the requested data, it returns as many whole instances of the designated **byte stream type** as it finds.

Reading Datalog Files

If **refnum** is a datalog file refnum, the Read File function reads records from the datalog file specified by **refnum**. If the data in the file does not match the datatype associated with the datalog file, this function returns an error.

The number of records read can be less than specified by **count** if this function encounters the end of the file. The function sets the file mark to the record following the last record read. (You should never encounter a partial record; if you do, the file is corrupt.)

Do not wire **convert eol**, **line mode**, and **byte stream type**. They do not pertain to datalog files. The **count** and **data** parameters function in the following manner.

count is the number of records to read and may be wired or unwired. If you do not wire **count**, the

function returns a single record of the datalog type specified when the file is created or opened. For example, if the type is a 16-bit integer, the function returns one 16-bit integer. If the type is an array of 16-bit integers, the functions returns one array of 16-bit integers. (Your records typically consist of clusters of diverse elements, but the rules for simple types used in these examples apply to those as well.)

If you wire **count**, it can be a scalar number, in which case the function returns a 1D array of records. Or it can be a cluster of N scalar numbers, in which case the function returns an N-dimension array of records.

If the wired **count** is a scalar number, and the datalog type is something other than an array, the function returns that number of records in a 1D array. For example, if the type is a single-precision, floating-point number and **count** is 3, the array contains three, single-precision, floating-point numbers. However, if the type is an array, the function returns the records in a cluster array (because LabVIEW does not have arrays of arrays). Therefore, if the datalog type is an array of single-precision, floating-point numbers and **count** is 3, the function returns a cluster array of three, single-precision, floating-point number arrays.

If the wired **count** is a cluster of N numbers, the function returns an N-dimension array of records. The size of each dimension is the value of the corresponding number according to its cluster order. The number of records returned in this manner is the product of the N numbers. Therefore, you can return 20 records as a 2D array of two columns and ten rows by wiring a two-element cluster with element 0 = 2 and element 1 = 10 to **count**.

Wiring byte stream type

When you wire **byte stream type**, **byte stream type**, **count**, and **data** function in the manner described in the following paragraphs. In this case, do not wire **line mode** or **convert eol**.


byte stream type can be any datatype. Read File interprets the data starting at the designated byte offset to be **count** instances of that type. If the type is variable-length, that is, an array, a string, or a cluster containing an array or string, the function assumes that each instance of the type contains the length or dimensions of that instance. If they do not, the function misinterprets the data. If LabVIEW determines that the data does not match the type, it sets the value of **data** to the default value for its type and returns an error.

count is the number of instances of the **byte stream type** to read. If **count** is unwired, the function returns a single instance of the **byte stream type**.
If you wire **count**, it can be a scalar number, in which case the function returns a 1-D array of instances of the **byte stream type**. Or it can be a cluster of N scalar numbers, in which case the function returns an N-dimension array of instances of the **byte stream type**.

If the wired **count** is a scalar number and the **byte stream type** is something other than an array, the function returns that number of instances in a 1D array. For example, if you leave type unwired, and **count** is 3, the functions returns a string with three characters. If the type is a single-precision, floating point number, the function returns an array of three, single-precision, floating point numbers. However, if the type is an array, the function returns the instances in a cluster array, because LabVIEW does not have arrays of arrays. Therefore, if the type is an array of single-precision, floating point numbers and **count** is 3, the function returns a cluster array of three, single-precision, floating point number arrays.

If the wired **count** is a cluster of N numbers, the function returns an N-dimension array of instances of the type. The size of each dimension is the value of the corresponding number according to its cluster order. The number of instances returned in this manner is the product of the N numbers. Thus, you can return 20, single-precision, floating point numbers as a 2D array of two columns and ten rows by wiring a two-element cluster with element 0 =2 and element 1 = 10 to **count**.


data contains the data read from the file. Refer to the previous description of **count** for an explanation of the structures data can have.

When You Do Not Wire Byte Stream Type

When you do not wire **byte stream type**, the **line mode**, **count**, **convert eol**, and **data** parameters function in the manner described below.

line mode, in conjunction with **count**, determines when the read stops.
If **line mode** is TRUE, and if you do not wire **count** or **count** equals 0, Read File reads until it encounters an end of line marker--a carriage return, a line feed, or a carriage return followed by a line feed, or it encounters end of file. If **line mode** is TRUE, and **count** is greater than 0, Read File reads until it encounters an end of line marker, it encounters end of file, or it reads **count** characters.

If **line mode** is FALSE, Read File reads **count** characters. In this case, if you do not wire **count**, it defaults to 0. **line mode** defaults to FALSE.

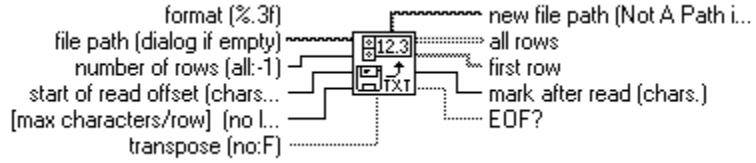
count is the number of characters to read. You can wire **count** only to a scalar numeric.

convert eol (F) determines whether the function converts the end of line markers it reads into LabVIEW end of line markers. The system-specific end of line marker is a carriage return followed by a line feed on Windows, a carriage return on Macintosh, and a line feed on UNIX. The LabVIEW end of line marker is a line feed.
If **convert eol** is TRUE, the function converts all end of line markers it encounters into line feeds.
If **convert eol** is FALSE, the function does not convert the end of line markers it reads. **convert eol** defaults to FALSE.

data is a string of characters read from the file.

Read From Spreadsheet File

Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D, single-precision array of numbers. You can optionally transpose the array. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format. This VI calls the Spreadsheet String to Array function to convert the data.



file path (dialog if empty) is the path name of the file. If file path is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

number of rows (all:-1) is the maximum number of rows or lines the VI reads. For this VI, a row is a character string ending with a carriage return, line feed, or a carriage return followed by a line feed; a string ending at the end of file; or a string that has the maximum line length specified by the max characters per row input. If **number of rows** is <0, the VI reads the entire file. The default value is -1.

format (%.3f) specifies how to convert the characters to numbers; the default is %.3f. Refer to the discussion of format strings and the Spreadsheet String To Array function.

delimiter (Tab) is an input string that you can use to format columns in spreadsheets. You can use tabs, commas, and so on for delimiters in your spreadsheet.

start of read offset (chars...) is the position in the file, measured in characters (or bytes), at which the VI begins reading.

[max characters/row] (no...1) is the maximum number of characters the VI reads before ending

the search for the end of a row or line. The default is 0, which means that there is no limit to the number of characters the VI reads.

transpose (no:F) Set TRUE to transpose the data after converting it from a string. The default value is FALSE.

all rows is the data read from the file in the form of a 2D array of single-precision numbers.

new file path (Not a Path i...) is the path of the file from which the VI reads data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

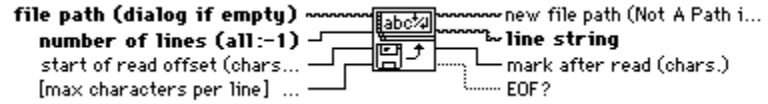
first row is the first row of the **all rows** array in the form of a 1D array of single-precision numbers. You can use this output when you want to read one row into a 1D array.

mark after read (chars.) is the location of the file mark after the read; it points to the character (byte) in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of the file.

Read Lines From File

Reads a specified number of lines from a byte stream file beginning at a specified character offset. The VI opens the file before reading from it and closes it afterwards.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

number of lines (all:-1) is the maximum number of lines the VI reads. For this VI, a line is a character string ending with a carriage return, line feed, or a carriage return followed by a line feed; a string ending at the end of file; or a string that has the maximum line length specified by the **max characters per line** input. If **number of lines** is <0, the VI reads the entire file. The default value is -1.

start of read offset (chars...) is the position in the file, measured in characters (or bytes), at which the VI begins reading.

[max characters per line] is the maximum number of characters the VI reads before ending the search for the end of a line. The default is 0, which means that there is no limit to the number of characters the VI reads.

line string is the data read from the file.

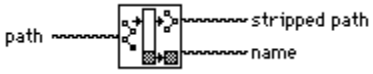
new file path (Not a Path i...) is the path of the file from which the VI reads data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

mark after read (chars.) is the location of the file mark after the read; it points to the character (byte) in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of the file.

Strip Path

Returns the **name** of the last component of a path and the **stripped path** that leads to that component.



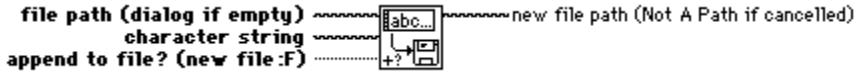
path specifies the path on which you want to operate. If this parameter is an empty path or is invalid, this function returns an empty string in **name** and Not A Path in **stripped path**.

stripped path is the resulting path obtained by removing **name** from the end of **path**.

name is the last component of the specified path.

Write Characters To File

Writes a character string to a new byte stream file or appends the string to an existing file. The VI opens or creates the file before writing to it and closes it afterwards.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

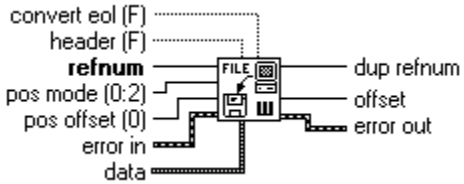
character string is the data the VI writes to the file.

append to file? (new file:F). Set to TRUE if you want to append the data to an existing file; you can also set it to TRUE to write to a new file. Set to FALSE (default value) if you want to write the data to a new file or to replace an existing file.

new file path (Not A Path if cancelled) is the path of the file to which the VI wrote data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

Write File

Writes data to the file specified by **refnum**. Writing begins at a location specified by **pos mode** and **pos offset** for byte stream file and at the end of file for datalog files. **data**, **header**, and the format of the specified file determine the amount of data written.



The **refnum**, **dup refnum**, **offset**, **error in,** and **error out** parameters function in the following manner for both datalog and byte stream files. The **convert eol (F)**, **header (F)**, **pos mode (0:2)**, **pos offset (0)**, and **data** parameters are discussed in the Byte Stream Files and Writing Datalog Files topics.

refnum is the file refnum associated with the file to which you want to write data.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

dup refnum is a flow-through parameter with the same value as **refnum**.

offset indicates the new location of the mark relative to the beginning of the file. **offset** is expressed in units of records for datalog files and bytes for byte stream files.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Byte Stream Files

If **refnum** is a byte stream file refnum, the Write File function writes to a location specified by **pos mode** and **pos offset** in the byte stream file specified by **refnum**. If the top-level datatype of **data** is of variable length (that is, a string or an array), Write File can write a header to the file that specifies the size of the

data. LabVIEW sets the file mark to the byte following the last byte written. **pos mode**, **pos offset**, **header**, **convert eol**, and **data** function in the following manner for byte stream files.

pos mode (0:2), together with **pos offset**, specifies where the write operation begins.

0: **pos offset** indicates that the write operation begins at the beginning of the file plus **pos offset**.
1: **pos offset** indicates that the write operation begins at the end of the file plus **pos offset**.
2: **pos offset** indicates that the write operation begins at the current location of the file mark plus the **pos offset**.

If you wire **pos offset**, **pos mode** defaults to 0, and the offset is relative to the beginning of the file. If you do not wire **pos offset**, **pos mode** defaults to 2, and the write operation starts at the current file mark.

pos offset (0) specifies how far from the location specified by **pos mode** to start writing. **pos offset** is expressed in terms of bytes and defaults to 0 if unwired.

header (F) can be wired only if the top-level datatype of **data** has variable length (that is, if **data** is a string or an array).

If **header** is TRUE, this function writes the top-level datatype of **data** with a header specifying its size. If **header** is FALSE, this function writes the top-level datatype of **data** without a header. LabVIEW always writes a variable-length datatype below the top-level (that is, contained in the top-level datatype) with a header specifying its size.

header defaults to FALSE.

convert eol (F) determines whether the function converts the end of line markers it writes into system-specific end of line markers. You can wire **convert eol** only if **data** is a string. The system-specific end of line marker is a carriage return followed by a line feed on Windows, a carriage return on Macintosh, and a line feed on UNIX. The LabVIEW end of line marker is a line feed.

If **convert eol** is TRUE, the function converts all end of line markers it writes into system-specific end of line markers. If **convert eol** is FALSE, the function does not convert the end of line markers it writes. **convert eol** defaults to FALSE. If **header** is TRUE, LabVIEW ignores **convert eol**.

data contains the data to write to the file. To perform character-oriented I/O, wire a string to **data** and leave **header** unwired, because it defaults to FALSE. The VI writes the characters of **data** in sequence without any header information.

Writing Datalog Files

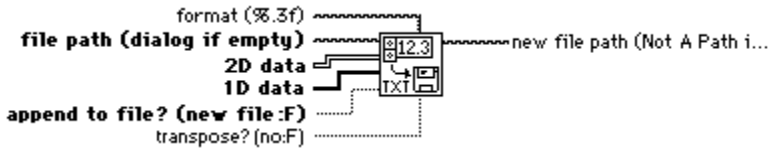
If **refnum** is a datalog file refnum, the Write File function writes data as records to the datalog file specified by **refnum**. Writing always starts at the end of the datalog file (datalog files are append-only). LabVIEW sets the file mark to the record following the last record written. The **convert eol**, **header**, **pos mode**, and **pos offset** parameters do not apply with datalog files, and you cannot wire them. The **data** parameter functions in the following manner for datalog files.

data must be either a datatype that matches the datatype specified when you open or create the file, or an array of such datatypes. In the former case, this function writes **data** as a single record in the datalog file. Representation of numeric data is coerced to the representation of the datatype if necessary. In the latter case, this function writes each element of **data** as a separate record in the datalog file in row-major order.

Write To Spreadsheet File

Converts a 2D or 1D array of single-precision (SGL) 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 before writing to it and closes it afterwards. You can use this VI to create a

text file readable by most spreadsheet applications. This VI calls the Array to Spreadsheet String function to convert the data.



file path (dialog if empty) is the path name of the file. If file path is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

2D data contains the single-precision numbers the VI writes to the file if **1D data** is not wired or is empty.

1D data contains the single-precision numbers the VI writes to the file if this input is not empty. The VI converts the 1D array into a 2D array before proceeding. If **transpose?** is FALSE, each call to this VI creates a new line or row in the file.

append to file? (new file:F). Set to TRUE if you want to append the data to an existing file; you can also set it to TRUE to write to a new file. Set to FALSE (default value) if you want to write the data to a new file or to replace an existing file.

format (%.3f) specifies how to convert the numbers to characters. If the format string is %.3f (default), the VI creates a string long enough to contain the number, with three digits to the right of the decimal point. If the format is %d, the VI converts the data to integer form using as many characters as necessary to contain the entire number. Refer to the discussion of format strings and the Array To Spreadsheet String function.

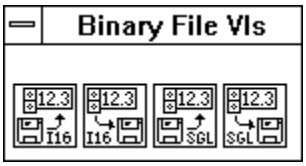
delimiter (Tab) is an input string that you can use to format columns in spreadsheets. You can use tabs, commas, and so on for delimiters in your spreadsheet.

transpose? (no:F). Set to TRUE to transpose the data before converting it to a string. The default value is FALSE.

new file path (Not A Path i...) is the path of the file to which the VI wrote data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

Binary File VI Descriptions

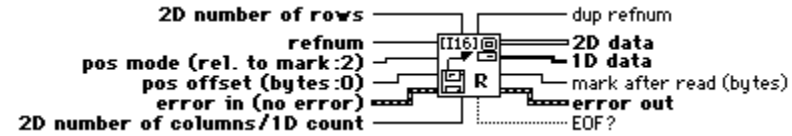
The following illustration displays the options available on the **Binary File VIs** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Read From I16 File
Read From SGL File
Write To I16 File
Write To SGL File

Read From I16 File

Reads a 2D or 1D array of data from a byte stream file of signed, word integers (I16). The VI opens the file before reading from it and closes it afterwards. You can use this VI to read unscaled or binary data acquired from data acquisition VIs and written to a file with Write To I16 File.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

2D number of rows is the number of rows to create when the data returns in a 2D array. The default value is 0. See the following Note.

2D number of columns/1D count is the number of columns to create if the data returns in a **2D array**, or it is the number of elements to return into **1D array**. The default value is -1. See the following Note.

Note: **To read the entire file into 1D array, set 1D count <0 (default); you can leave 2D number of rows unwired. This is the default operation if both inputs are unwired.**

To read N elements into 1D array, set 1D count to N and leave 2D number of rows unwired or set to 0.

To read the entire file into 2D array, set 2D number of rows <0 and 2D number of columns = N > 0. The VI calculates the number of rows as the integer part of file size/N.

To read M rows or N columns each into 2D array, set 2D number of rows = M and 2D number of columns = N.

The following table displays the information in this Note.

2D Number of Columns/1D Count	2D Number of Rows	Output
<0 (default)	value does not matter	entire file into 1D array

	(default)	
N>0	0	N numbers into 1D array
N>0	<0	entire file into 2D array
N>0	M>0	NxM numbers into 2D array

Other combinations result in empty arrays.

start of read offset (bytes:0) is the position in the file, measured in bytes, at which the VI begins to read. The offset unit is bytes rather than numbers because byte-stream files can contain segments of different types of data. Therefore, to read an array of 100 numbers that follows a header of 57 characters, set the **start of read offset** to 57.

2D array contains the 16-bit numbers read from the file if **2D number of rows** and **2D number of columns** define a **2D array**; otherwise, this output is empty.

1D array contains the 16-bit numbers read from the file if **1D count** and **2D number of rows** define a **1D array**; otherwise, this output is empty.

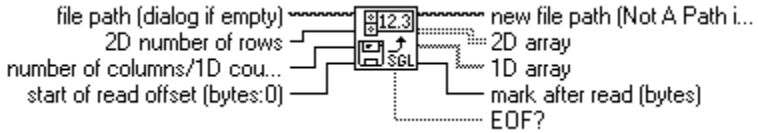
new file path (Not A Path if...) is the path of the file from which the VI reads data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

mark after read (bytes) is the location of the file mark after the read; it points to the byte in the file following the last byte read.

EOF? is TRUE if you attempt to read past the end of file.

Read From SGL File

Reads a 2D or 1D array of data from a byte stream file of single-precision numbers (SGL). The VI opens the file before reading from it and closes it afterwards. You can use this VI to read scaled data acquired from data acquisition VIs and written to a file with Write To SGL File.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

2D number of rows is the number of rows to create when the data is returned into a **2D array**. The default value is 0. See the following Note.

2D number of columns/1D count is the number of columns to create if the data is returned into **2D array**, or it is the number of elements to return into **1D array**. The default value is -1. See the following Note.

Note: To read the entire file into 1D array, set 1D count <0 (default); you can leave 2D number of rows unwired. This is the default operation if both inputs are unwired.

To read N elements into 1D array, set 1D count to N and leave 2D number of rows unwired or set to 0.

To read the entire file into 2D array, set 2D number of rows <0 and 2D number of columns =

N > 0. The VI calculates the number of rows as the integer part of file size/N.

To read M rows or N columns each into 2D array, set 2D number of rows = M and 2D number of columns = N.

The following table displays the information in this Note.

2D Number of Columns/1D Count	2D Number of Rows	Output
<0 (default)	donÕt care	entire file into 1D array
N>0	0	N numbers into 1D array
N>0	<0	entire file into 2D array
N>0	M>0	NxM numbers into 2D array

Other combinations result in empty arrays.

start of read offset (bytes:0) is the position in the file, measured in bytes, at which the VI begins reading. The offset unit is bytes rather than numbers because byte-stream files can contain segments of different types of data. Therefore, to read an array of 100 numbers that follows a header of 57 characters, set the **start of read offset** to 57.

new file path (Not A Path i...) is the path of the file from which the VI read data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

2D array contains the single-precision numbers read from the file if **2D number of rows** and **2D number of columns** define a **2D array**; otherwise, this output is empty.

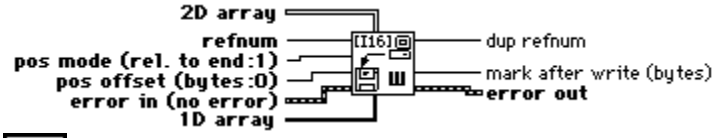
1D array contains the single-precision numbers read from the file if **2D number of rows** and **1D count** define a **1D array**; otherwise, this output is empty.

mark after read (bytes) is the location of the file mark after the read; it points to the byte in the file following the last byte read.

EOF? is TRUE if you attempt to read past the end of file.

Write To I16 File

Writes a 2D or 1D array of signed word integers (I16) to a new, byte stream file or appends the data to an existing file. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to write unscaled or binary data from data acquisition VIs.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

2D array contains the 16-bit numbers that the VI writes to the file if **1D array** is not wired or empty.

1D data contains the 16-bit numbers that the VI writes to the file if this input is not empty.

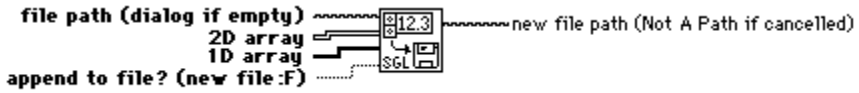
append to file? (new file:F). Set to TRUE if you want to append the data to an existing file; you can also set it to TRUE to write to a new file. Set to FALSE (default value) if you want to write the data to

a new file or to replace an existing file.

new file path (Not A Path i...) is the path of the file to which the VI wrote data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

Write To SGL File

Writes a 2D or 1D array of single-precision numbers (SGL) to a new byte stream file or appends the data to an existing file. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to write scaled data from data acquisition VIs without changing the representation.



file path (dialog if empty) is the path name of the file. If **file path** is empty (default value) or is Not A Path, the VI displays a file dialog box from which you can select a file. Error 43 occurs if you select **Cancel** from the dialog box.

2D array contains the single-precision numbers the VI writes to the file if **1D array** is not wired or is empty.

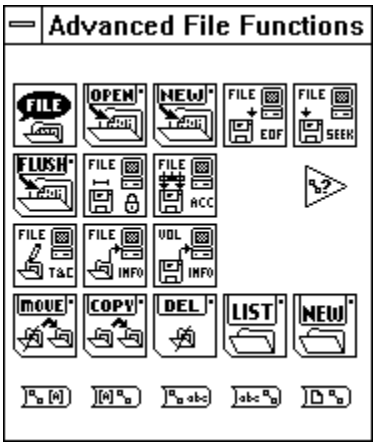
1D array contains the single-precision numbers the VI writes to the file if this input is not empty.

append to file? (new file:F) Set to TRUE if you want to append the data to an existing file; you can also set it to TRUE to write to a new file. Set to FALSE (default value) if you want to write the data to a new file or to replace an existing file.

new file path (Not A Path i...) is the path of the file to which the VI wrote data. You can use this output to determine the path of a file that you open using a file dialog box. **new file path** returns Not A Path if you select **Cancel** from the dialog box.

Advanced File Function Descriptions

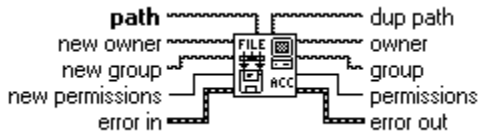
The following illustration displays the options available on the **Advanced File Functions** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Access Rights
Array Of Strings To Path
Copy
Delete
EOF
File Dialog
File/Directory Info
Flush File
List Directory
Lock Range
Move
New Directory
New File
Open File
Path To Array Of Strings
Path To String
Path Type
Refnum To Path
Seek
String To Path
Type and Creator
Volume Info

Access Rights

Sets and returns the owner, group, and permissions of the file or directory specified by **path**. If you do not specify **new owner**, **new group**, or **new permissions**, this function returns the current settings unchanged.



(Windows) The Access Rights function ignores **new owner** and **new group** and returns empty strings for **owner** and **group** because Windows does not support owners and groups.

(Macintosh) If path refers to a file, the Access Rights function ignores **new owner** and **new group** and returns empty strings for **owner** and **group** because Macintosh does not support owners or groups for files.

path specifies the file or directory whose access rights you want to change.

new owner specifies the new owner setting for the file or directory.

new group specifies the new group setting for the file or directory.

new permissions specifies the new permissions setting for the file or directory. See the Permissions topic for a detailed descriptions of this input.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

dup path is a flow-through parameter with the same value as **path**.

owner contains the current owner setting for the file or directory after this function executes.

group contains the current group setting for the file or directory after this function executes.

permissions contains the current permissions setting for the file or directory after this function executes. See the Permissions topic, for more information about this output.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Array Of Strings To Path

Converts an **array of strings** into a relative or absolute **path**.

array of strings contains the names of the components of the path you want to build. The first element is the highest level of the path hierarchy (the volume name, for file systems that support multiple volumes), and the last element is the last element of the hierarchy. An element that contains an empty string tells LabVIEW to go up a level in the hierarchy.

relative indicates whether you want to create a **relative** path or an absolute path. If you set **relative** to TRUE, it is a relative path; if set to FALSE, it is an absolute path. If you set **relative** to FALSE, and the path sepcified is not valid as an absolute path (for example, the path means *go up a level*), the function sets **path** to Not A Path.

path is the resulting path.

Copy

Copies the file or directory specified by **source path** to the location specified by **target path**. If you copy a directory, this function copies all its contents recursively.

source path specifies the file or directory you want to copy.

target path is the new path, including the new file or directory name, for the file or directory you want to copy. If a file already exists at **target path**, this function returns an error. If a directory exists at **target path**, this function tries to copy the file or directory at **source path** to the location specified by

appending the file or directory name (that is, the last element) of **source path** to **target path**. If a file or directory exists at this location, this function returns an error. This function also returns an error if the target path is an invalid path (for example, when the specified parent directory is not valid).

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

new path specifies the new location of the file or directory if the copy is successful. If not, this function sets **new path** to Not A Path.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Delete

Deletes the file or directory specified by **path**. If **path** specifies a directory that is not empty or if you do not have write permission for both the file or directory specified by **path** and its parent directory, this function does not remove the directory and returns an error.

path specifies the file or directory you want to delete.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

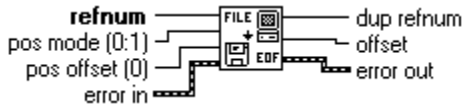
source identifies where the error occurred.

dup path is a flow-through parameter with the same value as **path**.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

EOF

Sets or returns the logical EOF (end-of-file) of the file identified by **refnum**. **pos mode** and **pos offset** specify the new location of the EOF. This function always returns the location of the EOF relative to the beginning of the file.



You cannot set the EOF of a datalog file. If **refnum** identifies a datalog file, you cannot wire **pos mode** and **pos offset**.

refnum identifies the file whose EOF you want to set.

pos mode (0:1), together with **pos offset**, specifies where to set the EOF.

0: **pos offset** indicates that EOF should be set to the beginning of the file plus **pos offset**.
1: **pos offset** indicates that EOF should be set to the end of the file plus **pos offset**.
2: **pos offset** indicates that EOF should be set to the current location of the file mark plus **pos offset**.

If the computed location does not exist in the file, for example, **pos mod** = 0 and **pos offset** =

-10, EOF is not set, and the function returns an error.

If you wire **pos offset**, **pos mode** defaults to 0, and the new location of the EOF is relative to the beginning of the file. If you do not wire **pos offset**, **pos mode** defaults to 1, and the location of the EOF is left unchanged.

pos offset (0) specifies how far from the location specified by **pos mode** to set the EOF. **pos offset** is expressed in terms of bytes and defaults to 0 if unwired.

Setting the new EOF to an offset smaller than the current logical EOF discards data at the end of the file. Setting the new EOF to an offset larger than the current EOF allocates additional space at the end of the file, although the data in this space is undefined.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

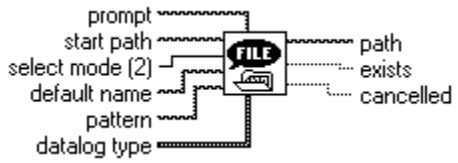
dup refnum is a flow-through parameter with the same value as **refnum**.

offset indicates the new location of the EOF relative to the beginning of the file. **offset** has units in records for datalog files and bytes for byte stream files.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

File Dialog

Displays a dialog box with which you can specify the path to a file or directory. You can use this dialog box to select existing files or directories or to select a location and name for a new file or directory.



select mode (2) indicates the types of files or directories you can specify using the dialog box. **select mode** defaults to 2 if you do not wire it.

0: Select only an existing file. You might use this value if you want the user to select the name of a file to open; the program can then open the file using the Open File function.

1: Select only a file that does not exist. You might use this value if you want the user to select the name of a file to create; the program can then create the file using the New File function.

2: Select either an existing or a non-existent file. You might use this value if you want the user to select the name of a file to create or append data to. This is the default if you leave **select mode** unwired.

3: Select only an existing directory. You might use this value if you want the user to select the name of a directory that contains data files. The program can then access those files with the directory path.

4: Select only a non-existent directory. You might use this value if you want the user to select the name of a directory that the program subsequently creates using the New Directory function.

5: Select either an existing or a non-existent directory. You might use this value if you want the user to select the name of a directory in which to store data files. The program can create the directory if it does not exist.

prompt is the message that appears below the list of files and directories in the dialog box.

prompt defaults to an empty string if unwired.

start path is the path of the directory whose contents LabVIEW initially displays in the dialog box. If **start path** is valid, but does not refer to an existing directory, LabVIEW strips names from the end of the path until the path is a valid directory path or an empty path. If **start path** is invalid or unwired, the last directory viewed in a file dialog initially appears in the dialog box.

default name is the name you want to appear as the initial file or directory name in the dialog box. This parameter defaults to the empty string if unwired.

pattern optionally restricts the files displayed in the dialog box to those whose name matches **pattern**. **pattern** does not restrict the directories displayed in the dialog box. The pattern matching performed here is similar to the one used in matching wildcards in DOS and UNIX filenames and not like the regular expression matching performed by the Match Pattern function. If you specify characters other than the question mark character (?) or the asterisk character (*), the File Dialog function displays only files or directories that contain those characters. You can use the question mark character (?) to match any single character. You can use the asterisk character (*) to match any sequence of one or more characters.

datalog type can be wired to any datatype. If wired, it restricts the files displayed in the dialog box to datalog files containing records of the specified datatype.

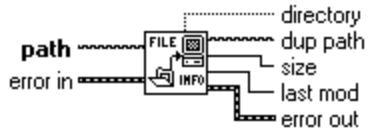
path is the full path to the file or directory selected using this dialog box. If you select Cancel, this function sets **path** to Not A Path**.**

exists is TRUE if **path** specifies an existing file or directory.

cancelled is TRUE if you close the dialog box by selecting **Cancel** or if an error occurs during the execution of the dialog box.

File/Directory Info

Returns information about the file or directory specified by **path**, including its **size**, its last modification date, and whether it is a directory.



path identifies the file or directory whose attributes you want to determine.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

directory is TRUE if the path points to a directory, and FALSE otherwise.

dup path is a flow-through parameter with the same value as **path**.

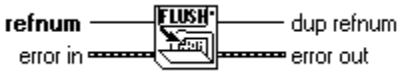
size indicates the size of the file or directory specified by **path**. If **path** specifies a directory, **size** indicates the number of items in the directory. Otherwise, **size** indicates the length in bytes of the specified file, whether the file is a datalog file or a byte stream file.

last mod indicates the date and time at which the file or directory was last modified in seconds since the standard reference time, 12:00 a.m., January 1, 1904, Universal Time.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Flush File

Writes all buffers of the file identified by **refnum** to disk and updates the directory entry of the file associated with **refnum**. The file remains open, and **refnum** remains valid.



Data written to a file often resides in a buffer until the buffer fills up or until you close the file. This function forces the operating system to write any buffer data to the file.

refnum is the file refnum associated with the file you want to flush.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

dup refnum is a flow-through parameter with the same value as **refnum**.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

List Directory

Returns two arrays of strings listing the names of all files and directories found in **directory path**, filtering both arrays based upon **pattern** and filtering the **file names** array based upon the specified **datalog type**.

directory path identifies the directory whose contents you want to determine. If this is not an existing directory, this function sets **file names** and **directory names** to empty arrays and returns an error.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

pattern restricts the files and directories returned to those whose names match **pattern**. The pattern matching this function performs is like that used in matching wildcards in DOS and UNIX filenames and is not like the regular expression matching performed by the Match Pattern function. If you specify characters other than the question mark character (€?€) or the asterisk character (€*€), this function lists only files and directories that contain those characters. You can use the question mark character (?€) to match any single character. You can use the asterisk character (€*€) to match any sequence of one or more characters.

datalog type can be any datatype. If you wire **datalog type**, it restricts the **file names** returned to only datalog files containing records of the specified datatype.

dup directory path is a flow-through parameter with the same value as **directory path**.

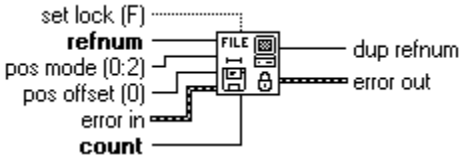
file names contains the names of the files found in the specified directory.

directory names contains the names of the directories found in the specified directory.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Lock Range

Locks or unlocks a range of a file specified by **refnum**.



Note: You cannot lock a range of a datalog file.

refnum is the file refnum associated with the file you want to lock.

count is the number of bytes this function locks or unlocks. You can lock or unlock a range that extends beyond the end of the file to ensure that no other users append to a file while you append to it. This function returns an error if **count** is less than zero.

pos mode (0:2), together with **pos offset**, specifies where the lock or unlock operation should begin.

0: **pos offset** indicates that the lock or unlock operation should begin at the beginning of the file plus **pos offset**.

1: **pos offset** indicates that the lock or unlock operation should begin at the end of the file plus **pos offset**.

2: **pos offset** indicates that the lock or unlock operation should begin at the current location of the file mark plus **pos offset**.

If the computed location does not exist in the file, for example, **pos mod** = 0 and **pos offset** = -10, no range is locked or unlocked, and the function returns an error.

If you wire **pos offset**, **pos mode** defaults to 0, and the starting location of the lock is relative to the beginning of the file. If you do not wire **pos offset**, **pos mode** defaults to 2, and the lock begins at the current file mark.

pos offset (0) specifies how far from the location specified by **pos mode** the lock begins. **pos offset** is expressed in terms of bytes and defaults to 0 if unwired.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

set lock (F) determines whether to lock or unlock the specified range. **set lock** defaults to FALSE.

If **set lock** is TRUE, this function locks the specified range of data in the specified file. If two locked ranges overlap, this function treats them as a single locked range.

If **set lock** is FALSE, this function unlocks the specified range of data in the specified file. If unlocking a subrange of a locked range leaves locked data at either end of the original locked range, this function treats both sets of data as separate locked ranges.

dup refnum is a flow-through parameter with the same value as **refnum**.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

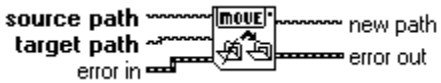
Locking a range of a file prevents both reading and writing by other users, overriding permissions for the file, and the deny mode associated with **refnum**. See Permissions for a full discussion of permissions.

Unlocking a range of a file removes the override caused by locking a range, so that the file's permissions and the deny mode associated with **refnum** determine whether other users can read from or write to that

range of the file.

Move

Moves the file or directory specified by **source path** to the location specified by **target path**.



source path is a path to the file or directory you want to move.

target path is the new path, including the new file or directory name, for the file or directory you want to move. If a file already exists at **target path**, this function returns an error. If a directory exists at **target path**, this function tries to move the file or directory at **source path** to the location specified by appending the file or directory name (that is, the last element) of **source path** to **target path**. If a file or directory exists at this location, this function returns an error. This function also returns an error if the target path is an invalid path (for example, when the specified parent directory is not valid).

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

new path specifies the new location of the file or directory if the move is successful. If not, this function sets **new path** to Not A Path.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

New Directory

Programmatically creates the directory specified by **directory path**. If a file or directory already exists at the specified location, this function returns an error instead of overwriting the existing file or directory.

directory path is the full path of the directory you want to create.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If status is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

group specifies the file system group you want to associate with the new directory. This parameter defaults to the system-specific default group, if unwired.

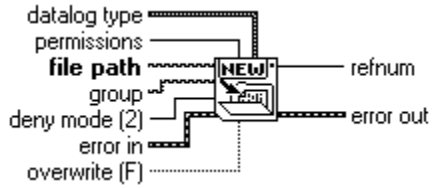
permissions specifies the file system access permissions to assign to the new directory. This parameter defaults to the system-specific default permissions, if unwired. See the Permissions topic for a detailed descriptions of this input.

dup directory path is a flow-through parameter with the same value as **directory path**.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

New File

Creates the file specified by **file path** and opens it for reading and writing (regardless of **permissions**).



file path is the full path of the file you want to create. If **file path** refers to an existing file or directory and overwrite is FALSE, this function does not create a new file. Instead, the function sets **refnum** to Not A Refnum and returns an error.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

overwrite (F). If you specify a path that already exists, **overwrite** determines whether the function replaces the file or returns an error. If **overwrite** is FALSE, the function returns a duplicate path error. If **overwrite** is TRUE, the function replaces the file. **overwrite** defaults to FALSE.

group specifies the file system group to be associated with the new file. If unwired, this parameter defaults to the system-specific default group.

deny mode (2) specifies the degree to which other users can operate on the file simultaneously. **deny mode** defaults to 2 if you do not wire it.

0: Deny both read and write to the file by other users.
1: Permit read but deny write to the file by other users.
2: Permit both read and write to the file by other users.

datalog type can be any datatype. If you wire it, LabVIEW creates a datalog file whose records are of the specified datatype, in which case the **refnum** output by New File is a datalog file refnum. If you leave **datalog type** unwired, LabVIEW creates a byte stream file, in which case the **refnum** output by New File is a byte stream file refnum.

permissions specifies the file system access permissions to assign to the new file. If unwired, this parameter defaults to the system-specific default permissions. See the Permissions topic for a detailed descriptions of this input.

refnum is the file refnum associated with the opened file. If **file path** refers to an existing file or directory and **overwrite** is FALSE, this function sets **refnum** to not-a-refnum.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Open File

Opens the file specified by **file path** for reading and/or writing.

file path is the full path of the file you want to open.

open mode (0) specifies whether this function opens the file for both reading and writing data, for reading only, or for writing only. **open mode** defaults to 0 if you do not wire it.

0: May both read and write to file.
1: May only read from file.

2: May only write to file. Does not truncate (remove all data from) the file. Under any non-Macintosh platform, this mode acts like mode 0.
3: May only write to file. Truncates the file.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

deny mode (2) specifies the degree to which other users can operate on the file simultaneously. **deny mode** defaults to 2 if you do not wire it.
0: Deny both read and write to the file by other users.
1: Permit read but deny write to the file by other users.
2: Permit both read and write to the file by other users.

datalog type can be wired to any datatype. If you wire it, LabVIEW assumes the specified file is a datalog file whose records are of the specified datatype, in which case the **refnum** output by Open File is a datalog file refnum. In this case, the function returns an error if the format of the file does not match the format for a datalog file whose records are of the specified datatype. If you leave **datalog type** unwired, LabVIEW assumes the specified file is a byte stream file, in which case the **refnum** output by Open File is a byte stream file refnum.

refnum is the file refnum associated with the opened file. If **file path** does not refer to an existing file, or if it refers to an existing file that you cannot open with the specified **open mode** and **deny mode**, this function sets **refnum** to not-a-refnum.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Path To Array Of Strings

Converts a **path** into an **array of strings** and indicates whether the path is **relative**.

path is the path you want to convert to an array of strings. If **path** is not-a-path, the **array of strings** is empty and **relative** is FALSE.

relative indicates whether you want to create a **relative** path or an absolute path. If you set **relative** to TRUE, it is a relative path; if set to FALSE, it is an absolute path. If you set **relative** to FALSE, and the path sepcified is not valid as an absolute path (for example, the path means *go up a level*), the function sets **path** to not a path.

array of strings contains the components of the path. The first element is the first step of the path hierarchy (the volume name, for file systems that support multiple volumes), and the last element is the file or directory specified by the path.

Path To String

Converts **path** into a string describing a path in the standard format of the platform.

path is the path you want to convert to a string. If **path** is not-a-path, the function sets **string** to not a path.

string is the path description in the standard format for the current platform of **path**. **string** is of the same data structure as **path**.

Path Type

Returns the type of the specified path, indicating whether it is an absolute, relative, or invalid path.

path specifies the path upon which you want to operate.

type is the type of the specified path.

0: Path is absolute.
1: Path is relative.
2: Path is invalid (either Not A Path or a badly formed path).

Refnum To Path

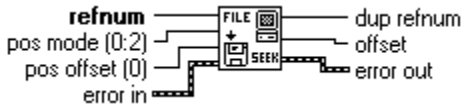
Returns the **path** associated with the specified **refnum.**

refnum is the refnum of an open file whose associated path you want to determine. If **refnum** is not a valid refnum, this function sets **path** to Not A Path, meaning the file is closed.

path is the corresponding path.

Seek

Moves the current file mark of the file identified by **refnum** to the position indicated by **pos offset** according to the mode chosen by **pos mode**.



refnum specifies the file whose current file mark you want to position.

pos mode (0:2) , together with **pos offset**, specifies where to move the current file mark.

0: **pos offset** indicates that the file mark should move to the beginning of the file plus **pos offset**.
1: **pos offset** indicates that the file mark should move to the end of the file plus **pos offset**.
2: **pos offset** indicates that the file mark should move to the current location of the file mark plus **pos offset**.

If the computed location does not exist in the file, for example,
pos mod = 0 and **pos offset** = -10, the file mark does not move, and the function returns an error.

If you wire **pos offset**, **pos mode** defaults to 0, and the offset is relative to the beginning of the file. If you do not wire **pos offset**, **pos mode** defaults to 2, and the location of the current file mark does not change.

pos offset (0) specifies how far from the location specified by **pos mode** to position the current file mark. If the file specified by **refnum** is a datalog file, **pos offset** is expressed in terms of records of the datatype stored in the datalog file; otherwise, **pos offset** is expressed in terms of bytes. **pos offset** defaults to 0 if unwired.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

dup refnum is a flow-through parameter with the same value as **refnum**.

offset indicates the new location of the current file mark relative to the beginning of the file. **offset** is expressed in the same terms as **pos offset** (records for datalog files and bytes for byte stream files).

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

String To Path

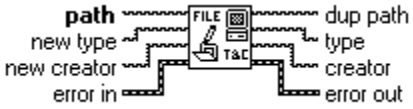
Converts a string, describing a path in the standard format for the current platform, to a path.

string can be a string, a cluster of strings, an array of strings, an array of clusters of strings, and so on. See the Polymorphism for String Conversion Functions topic for more information.

path is the platform-dependent representation of the path described by **string**. If **string** is not a valid path description on the current platform, the function sets **path** to not a path. **path** is of the same data structure as **string**.

Type and Creator

Reads and sets the type and creator of the file specified by **path**. File type and creator are four-character strings.



If no error occurs, this function returns the type and creator of the specified file in **type** and **creator**. If an error occurs, **error** is set to indicate the type of error, and **type** and **creator** are set to empty strings.

Note: **(Windows and UNIX) These systems do not support file types and creators. Trying to set the type or creator of a file in these platforms results in an error; however, you can get the file type and creator in these platforms. If the specified file has a name ending with characters that LabVIEW recognizes as specifying a file type (such as.**vi **for the LVIN file type and** *.llb* **for the LVAR file type), this function returns that type in type and** *LBVW* **in creator. Otherwise, the function returns** *????* **in both type and creator.**

path specifies the file or directory whose type or creator you want to set.

new type specifies the new type setting for the file. If you wire **new type**, this function sets the type of the specified file to **new type**. If you wire **new type** to a string other than four characters, an error occurs.

new creator specifies the new creator setting for the file. If you wire **new creator**, this function sets the creator of the specified file to **new creator**. If you wire **new creator** to a string other than four characters, an error occurs.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

dup path is a flow-through parameter with the same value as **path**.

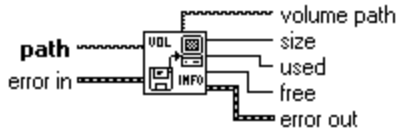
type contains the current type of the file after this function executes.If no error occurs, this function returns the type of the specified file in **type**. If an error occurs, **error** is set to indicate the type of error, and **type** is set to an empty string.

creator contains the current creator of the file after this function executes. If no error occurs, this function returns the type and creator of the specified file in **creator**. If an error occurs, **error** is set to indicate the type of error, and **creator** is set to an empty string.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

Volume Info

Returns information about the volume containing the file or directory specified by **path**, including the total storage space provided by the volume, the amount used, and the amount free in bytes.



path specifies the file or directory whose volume attributes you want to determine.

error in describes error conditions occurring before this VI executes. If an error has already occurred, this VI returns the value of the **error in** cluster in **error out**. The VI executes normally only if no incoming error exists; otherwise it merely passes the **error in** value to **error out**.

status is TRUE if an error occurred. If **status** is TRUE, this VI does not perform any operations.

code is the error code number identifying the error.

source identifies where the error occurred.

volume path is a new path that specifies the volume on which the specified file or directory resides.

size indicates the amount of storage, in bytes, provided by the specified volume.

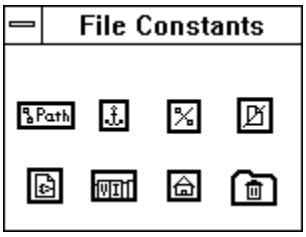
used indicates the amount of storage, in bytes, currently used on the specified volume.

free indicates the amount of storage, in bytes, available on the specified volume.

error out contains error information. If the **error in** cluster indicated an error, the **error out** cluster contains the same information. Otherwise, **error out** describes the error status of this VI.

File Constants Descriptions

The following illustration displays the options available on the **File Constants** subpalette. . Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Current VI's PATH
Default Directory
Empty Path
Not A Path
Not A Refnum
Path
Temporary Directory
VI Library

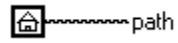
Current VI's Path Constant

Returns the path to the file containing the VI in which this function appears. If the VI is incorporated into an application (using the Application Builder libraries), the function returns the path to the VI in the application file, and treats the application file as a VI library.

path specifies the file of the current VI. If the VI has never been saved, path is Not A Path. The value of **path** always reflects the current location of the VI. If you move the VI, the value of this constant changes.

Default Directory Constant

Returns the path to your default directory. The default directory is the directory in which LabVIEW automatically stores information unless you specify otherwise. The LabVIEW **Preferences** dialog box, under **Paths**, defines this directory.



path specifies your default directory.

Empty Path Constant

Returns an empty path.

Not A Path Constant

Returns a path whose value is Not A Path. You can use this path as an output from structures and subVIs when an error occurs.

Not A Refnum Constant

Returns a refnum whose value is Not A Refnum. You can use this refnum as an output from structures and subVIs when an error occurs.

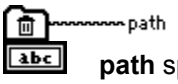
Path Constant

Use this to supply a constant directory or file path to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in the value. Use the standard file path syntax for a given platform.

The value of the path constant cannot be changed while the VI executes. You can assign a label to this constant.

Temporary Directory Constant

Returns the path to your temporary directory. The temporary directory is the directory in which you store information that you do not want to be stored to the default directory. The LabVIEW **Preferences** dialog box, under **Paths**, defines this directory.



path specifies your temporary directory.

VI Library Constant

Returns the path to the VI library directory for the current LabVIEW on the current computer. The LabVIEW Preferences dialog box (**Edit»Preferences**) defines this directory. If you build an application using the Application Builder libraries, this path is the path of the directory containing the application.

path specifies the VI library directory.

Build Path Function

Build Path

Close File Function

Close File

Open/Create/Replace File VI

Open/Create/Replace File

Read Characters From File VI

Read Characters From File

Read File Function

Read File

Read From Spreadsheet File VI

Read From Spreadsheet File

Read Lines From File VI

Read Lines From File

Strip Path Function

Strip Path

Write Characters To File VI

Write Characters To File

Write File Function

Write File

Write To Spreadsheet File.vi

Write To Spreadsheet File

Binary File VIs Subpalette

Binary File VIs

Advanced File Functions Subpalette

Advanced File Functions

File Constants Subpalette

File Constants

Read From I16 File VI

Read From I16 File

Read From SGL File VI

Read From SGL File

Write To I16 File VI

Write To I16 File

Write To SGL File VI

Write To SGL File

Access Rights Function

Access Rights

Copy Function

Copy

Delete Function

Delete

EOF Function

EOF

File Dialog Function

File Dialog

File/Directory Info Function

File/Directory Info

Flush File Function

Flush File

List Directory Function

List Directory

Lock Range Function

Lock Range

Move Function

Move

New Directory Function

New Directory

New File Function

New File

Open File Function

Open File

Path Type Function

Path Type

Seek Function

Seek

Type and Creator Function

Type and Creator

Volume Info Function

Volume Info

File I/O Function Overview

Click here to access the File I/O Function Descriptions topic.

General Behavior of File I/O Functions
UNC Filename Support of File I/O VIs
Byte Stream and Datalog Files
Flow-Through Parameters
Permissions
Error I/O in File I/O Functions

You can use the file VIs to write or read the following types of data.

- Strings to text files
- One-dimensional (1D) or two-dimensional (2D) arrays of single-precision numbers to spreadsheet text files.
- 1D or 2D arrays of single-precision or signed word integers to byte stream files.

The high-level file VIs described here call the low-level functions to perform complete, easy-to-use file operations. These VIs open or create a file, write or read to it, and close it. If an error occurs, these VIs display a dialog box that describes the problem and gives you the option to halt execution or to continue.

The high-level file VIs are located on the top row of the palette and consist of the following VIs:

Write Characters to File
Write to Spreadsheet File
Read Characters from File
Read from Spreadsheet File
Read Lines from File
Binary File VIslocated in the subpalette.

The low-level file functions perform one file operation at a time. These VIs perform error detection in addition to their other functions. The most commonly used low-level file functions are located on the second row of the palette. The remaining low-level functions are located in the **Advanced** subpalette.

General Behavior of File I/O Functions

The principal file operations are a three-step process. First, you create or open a file. Then you write data to the file or read data from the file. Finally, you close the file. Other file operations include creating directories; moving, copying, or deleting files; flushing files; listing directory contents; changing file characteristics; and manipulating paths.

When creating or opening a file, you must specify its location. Different computers describe the location of files in different ways, but most computer systems use a hierarchical system to specify the location of files. In a hierarchical file system, the computer system superimposes a hierarchy on the storage media. You can store files inside directories, which can contain other directories.

When you specify a file or directory in a hierarchical file system, you must indicate the name of the file or directory, as well as its location in the hierarchy. In addition, some file systems support the connection of multiple discrete media, called volumes. For example, DOS/Windows systems support multiple drives connected to a system; for most of these systems, you must include the name of the volume to create a complete specification for the location of a file. On other systems, such as UNIX, you do not need to specify the storage media locations for files because the operating system hides the physical implementation of the file system from you.

The method of identifying the target of a file function varies depending on whether the target is an open file. If the target is not an open file, or if it is a directory, you specify a target using the *path* of the target. The path describes the volume containing the target, the directories between the top-level and the target, and the name of the target. If the target is an open file, you use a *file refnum* to identify the file that LabVIEW is supposed to manipulate. The file refnum is an identifier that LabVIEW associates with the file when you open it. When you close the file, the file manager dissociates the file refnum from the file. In other words, the refnum is obsolete once the file is closed.

See, *Strings and File I/O*, Chapter 6 of the LabVIEW Tutorial Manual, and Path Controls and Refnums, for more information on path specification in LabVIEW and for file function examples.

UNC Filename Support of File I/O VIs for Win NT and Win '95

UNC filenames provide the primary means for specifying the location of a file or directory in a networked environment. In LabVIEW, you can enter and view UNC filenames in path controls, indicators, constants, and the file name box in the file dialog box. UNC filenames use the following path form.

\\<machine>\<share name>\<dir>\...\<file or dir>

where <machine> is the name of the computer that you want to access on the network, <share name> is the name of a shared drive on that machine, <dir>\... consists of the name of the directory and subdirectories in which you want to store the file, and <file> consists of the name that you want to call the file.

Byte Stream and Datalog Files

LabVIEW can make and access two types of files--byte stream and datalog files.

A *byte stream* file, as the name implies, is a file whose fundamental unit is a byte. A byte stream file can contain anything from a homogeneous set of one LabVIEW datatype to an arbitrary collection of datatypes--characters, numbers, Booleans, arrays, strings, clusters, and so on. An ASCII text file, a file containing this paragraph, for example, is perhaps the simplest byte stream file. A similar byte stream file is a basic spreadsheet text file, which consists of rows of ASCII numbers, with the numbers separated by tabs and the rows separated by carriage returns.

Another simple byte stream file is an array of binary 16-bit integers or single-precision, floating point numbers, which you acquire from a data acquisition (DAQ) program. A more complicated byte stream file is one in which an array of binary 16-bit integers or single-precision, floating point numbers is preceded by a header of ASCII text that describes how and when you acquired the data. That header could alternatively be a cluster of acquisition parameters, such as arrays of channels and scale factors, the scan rate, and so forth.

An Excel worksheet file, as opposed to an Excel text file, is also a more complicated form of byte stream file because it contains text interspersed with Excel-specific formatting data that does not make sense when you read it as text. In summary, you can make a byte stream file that consists of one each of all of LabVIEW datatypes. Byte stream files can be created using high-level VIs and low-level functions.

A *datalog* file, on the other hand, consists of a sequence of identically-structured records. Like byte stream files, the components of a datalog record can be any LabVIEW datatype. The difference is that all the datalog records must be the same type. Datalog files can only be created using low-level file functions.

You write a byte stream file typically by appending new strings, numbers, or arrays of numbers of any length to the file. You can also overwrite data anywhere within the file. You write a datalog file by appending one record at a time. You cannot overwrite the record.

You read a byte stream file by specifying the byte offset or index and the number of instances of the specified byte stream type you want to read. You read a datalog file by specifying the record offset or index and the number of records you want to read.

You use byte stream files typically for text or spreadsheet data that other applications may need to read. You can use byte stream files to record continuously acquired data that you need to read sequentially or randomly in arbitrary amounts. You use datalog files typically to record multiple test results or waveforms that you read one at a time and treat individually. Datalog files are difficult to read from non-LabVIEW applications.

Flow-Through Parameters

Many file functions contain *flow-through* parameters, which return the same value as an input parameter. You can use these parameters to control the execution order of the functions. By wiring the flow-through output of the first node you want to execute to the corresponding input of the next node you want to execute, you create artificial data dependency. Without these flow-through parameters, you would often have to use Sequence structures to ensure that file I/O operations take place in the correct order.

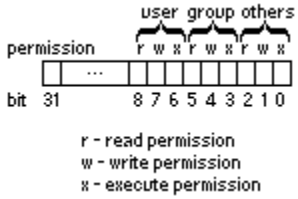
Permissions

Some of the File I/O VIs have a 32-bit integer parameter called **permissions** or **new permissions**. LabVIEW uses only the least significant nine bits of the 32-bit integer to determine file and directory access permissions.

(Windows) LabVIEW ignores the permissions for directories. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is read-only. Otherwise, you can write to the file.

(Macintosh) LabVIEW uses all 9 bits of permissions for directories. The bits which control read, write, and execute permissions, respectively, on a UNIX system are used to control See Files, Make Changes, and See Folders access rights, respectively, on the Macintosh. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is locked. Otherwise, the file is not locked.

(UNIX) The nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for users, groups, and others. The following illustration shows the permission bits on a UNIX system.



Error I/O in File I/O Functions

LabVIEW uses error I/O clusters, consisting of **error in** and **error out**, in all of its file I/O functions. With error I/O clusters you can string together several functions. When an error occurs in a function, that function does not execute and then passes the error along to the next function. When the error passes to subsequent functions, the subsequent function does not execute and passes the error along to the following function, and so on. The following illustration displays an example of the **error in** and **error out** clusters.

Note: Error I/O functions uniquely in the Close File function, which closes regardless of whether an error occurred in a preceding operation, insuring that files are closed correctly.

Although the error I/O clusters determine whether an error occurs, you must use error handlers to specify

how to handle those errors. For more information on error I/O, see the Error Handling in LabVIEW topic.

Array Of Strings To Path Function

Array Of Strings To Path

Path To Array Of Strings Function

Path To Array Of Strings

Path To String Function

Path To String

Refnum to Path Function

Refnum To Path

String To Path Function

String To Path

Introduction to LabVIEW

This topic contains the following subtopics:

What Is LabVIEW?
How Does LabVIEW Work?
Where Should I Start?
Organization of the LabVIEW System:
- Windows
- Macintosh
- UNIX

What Is LabVIEW?

LabVIEW is a program development application, much like C or BASIC, or 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:

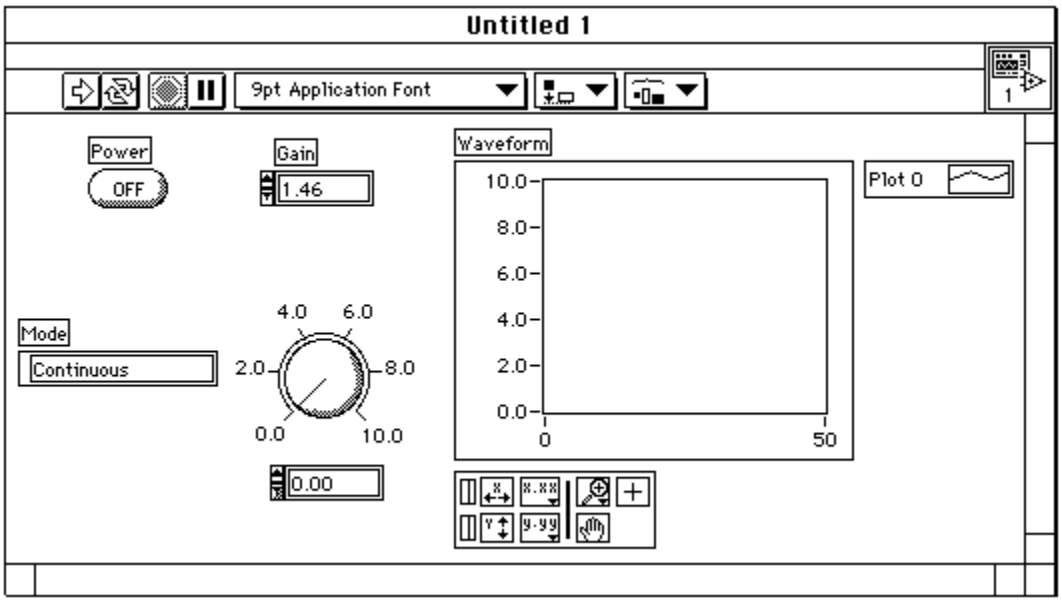
- The interactive user interface of a VI is called the *front panel*, because it simulates the panel of a physical instrument. The front panel can contain knobs, push buttons, graphs, and other controls and indicators. You enter data using a mouse and keyboard, and then view the results on the computer 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.
- The *icon and connector* of a VI work like a graphical parameter list so that other VIs can pass data to a subVI. The icon and connector allow you to use VIs as top-level programs, or as subprograms (*subVIs*) within other programs or subprograms.

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.

Front Panel

The user interface of a VI is like the user interface of a physical instrument--the front panel. A front panel of a VI might look like the following illustration.



The front panel of a VI is primarily a combination of *controls* and *indicators.* Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices that display data acquired or generated by the block diagram of the VI.

You add controls and indicators to the front panel by selecting them from the floating **Controls** palette shown in the following illustration.

You can change the size, shape, and position of a control or indicator. In addition, each control or indicator has a pop-up menu you can use to change various attributes or select different options. You access this pop-up menu (or pop up on this menu) by

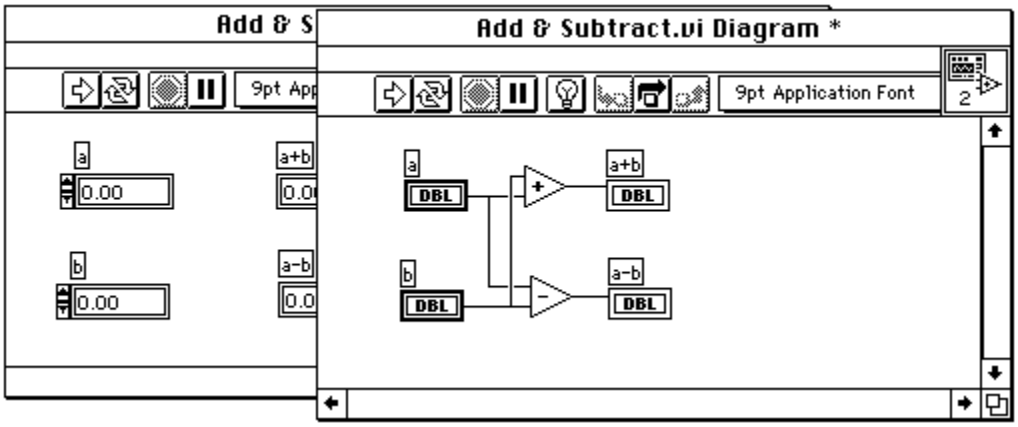
- ***(Windows and UNIX)*** clicking the object with the right mouse button
- ***(Macintosh)*** `<command>`-clicking

See Creating VIs, and topics in the Front Panel Reference for information on building a front panel.

Block Diagram

The diagram window holds the block diagram of the VI, which is the graphical source code of a LabVIEW VI. You construct the block diagram by *wiring* together objects that send or receive data, perform specific functions, and control the flow of execution.

The following simple VI computes the sum of and difference between two numbers. The diagram shows several primary block diagram program objects--*nodes*, *terminals*, and *wires*.



When you place a control or indicator on the front panel, LabVIEW places a corresponding terminal on the block diagram. You cannot delete a terminal that belongs to a control or indicator. The terminal disappears only when you delete its control or indicator.

The Add and Subtract function icons also have terminals. Think of terminals as entry and exit ports. Data that you enter into the controls (a and b) *exits* the front panel through the control terminals on the block diagram. The data then *enters* the Add and Subtract functions. When the Add and Subtract functions complete their internal calculations, they produce new data values at their *exit* terminals. The data flows to the indicator terminals and reenters the front panel, where it is displayed. The data exits from a *source terminal* and enters a destination or *sink terminal*.

Nodes are program execution elements. They are analogous to statements, operators, functions, and subroutines in conventional programming languages. The Add and Subtract functions are one type of node. LabVIEW has an extensive library of functions for math, comparison, conversion, I/O, and more. Another type of node is a *structure*. Structures are graphical representations of the loops and case statements of traditional programming languages, repeating blocks of code or executing them conditionally. LabVIEW also has special nodes for linking to external text-based code and for evaluating text-based formulas.

Wires are the data paths between source and sink (destination) terminals. You cannot wire a source terminal to another source nor can you wire a sink terminal to another sink. You can wire one source to several sinks. Each wire has a different style or color, depending on the data type that flows through the wire. The previous example shows the wire style for a numeric scalar value--a thin, solid line.

The principle that governs LabVIEW program execution is called *data flow*. Stated simply, a node executes only when all data inputs have arrived; the node supplies data to all of its output terminals when it finishes executing; and the data passes immediately from source to sink (or destination) terminals. Data flow contrasts with the control flow method of executing a conventional program, in which instructions are executed in the sequence in which they are written. Control flow execution is instruction driven. Dataflow execution is *data driven* or *data dependent*.

See topics in the Block Diagram Reference for in-depth information on using block diagram objects to build a program.

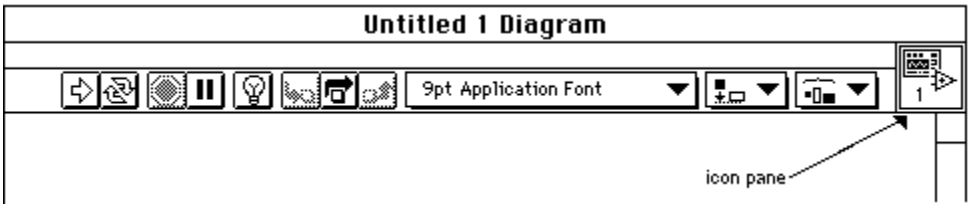
Icon and Connector

When an icon of a VI is placed on the diagram of another VI, it becomes a subVI, the LabVIEW version of a subroutine. The controls and indicators of a subVI receive data from and return data to the calling VIs diagram.

The *connector* is a set of terminals that correspond to the subVI controls and indicators. The icon is either the pictorial representation of the purpose of the VI, or a textual description of the VI or its terminals.

The connector is much like the parameter list of a function call; the connector terminals act like parameters. Each terminal corresponds to a particular control or indicator on the front panel. A connector receives data at its input terminals and passes the data to the subVI code via the subVI controls, or receives the results at its output terminals from the subVI indicators.

Every VI has a default icon, which is displayed in the icon pane in the upper right corner of the front panel and block diagram windows. This VI icon is shown in the following illustration.



Every VI also has a connector, which you access by choosing **Show Connector** from the icon pane pop-up menu on the front panel. When you show the connector for the first time, LabVIEW suggests a connector pattern. You can select a different pattern if you want. The connector generally has one terminal for each control or indicator on the front panel. You can assign up to 28 terminals. If you anticipate future changes to the VI that would require a new input or output, it is a good idea to leave some extra terminals unconnected. With the extra terminals, any new inputs or outputs will not affect other VIs that use this VI as a subVI.

These topics are more fully discussed in Creating SubVIs.

Where Should I Start?

If you are a new user of LabVIEW, or even if you are an experienced user, work through the *LabVIEW Tutorial* manual. It comprehensively describes the basics of how LabVIEW works. If you are upgrading from previous versions, you also may find new and useful information in the *LabVIEW Tutorial* because a number of new features have been added to LabVIEW.

Another good place to start is the Examples directory. Use the VI called `readme.vi`, at the top level of the directory, to browse through the available examples. When you select a VI, the online documentation for that VI is displayed (this information was entered for the VI using the Get Info... dialog box). To open a VI, use the **Open** command from the **File** menu.

(Windows, Macintosh, and Sun) If you are going to use data acquisition, you should read the *LabVIEW Data Acquisition Basics Manual* for data acquisition (DAQ) concepts and application examples. Then see the LabVIEW Data Acquisition VI topic for information about using the DAQ VIs.

The DAQ examples directory (`examples\daq\run_me.llb`) contains a VI library called `run_me` that has a getting started example VI for analog input, analog output, digital I/O, and counters. The *LabVIEW Data Acquisition Basics Manual* discusses these getting started example VIs and explains how the DAQ VIs work. Part 5, *SCXI--Getting Your Signals in Great Condition*, explains how to use the same example VIs with SCXI hardware. The *LabVIEW Data Acquisition Basics Manual* is an excellent starting place for data acquisition.

(Windows) Organization of the LabVIEW System

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.
- `labview.rsc`, `lvstring.rsc`, and `lvicon.rsc`--Data files used by the LabVIEW application
- `vi.lib` directory--Contains libraries of VIs that are included with LabVIEW, including GPIB, analysis, and data acquisition (DAQ) VIs.
- `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`.
- `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.

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*.

(Macintosh) Organization of the LabVIEW System

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.
- `lvstring.rsrc` and `lvicon.rsrc`--Data files used by the LabVIEW application.
- `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` folder--Contains numerous subfolders of examples. This folder also contains a VI called `readme.vi` that serves as a guide to the examples.

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 development system:files for 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*.LabVIEW development system:organization of (Macintosh)

(UNIX) Organization of the LabVIEW System

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.
- `lvstring.rsc` and `lvicon.rsc`--Data files used by the LabVIEW application
- `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`.

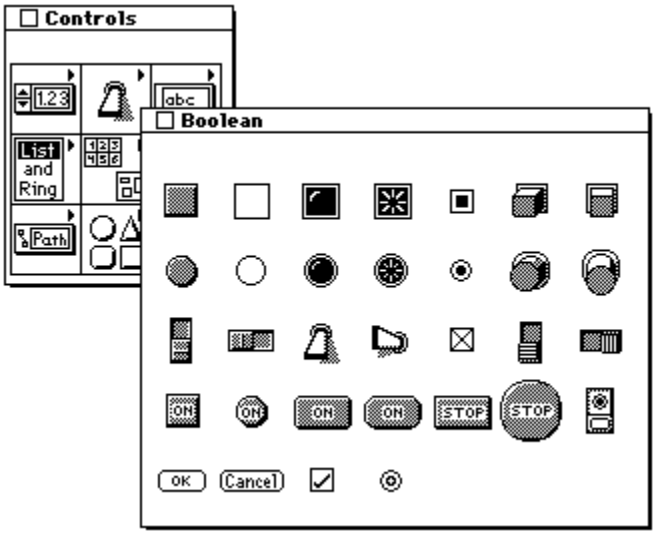
Boolean Controls and Indicators

This topic discusses how to create, operate, and configure Boolean controls and indicators.

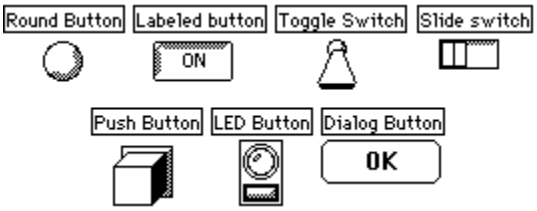
Creating and Operating Boolean Controls and Indicators
Configuring Boolean Controls and Indicators

Creating and Operating Boolean Controls and Indicators

A Boolean control or indicator has two values--TRUE or FALSE. Boolean controls and indicators are available from the **Boolean** palette in the **Controls** palette, as shown in the following illustration.

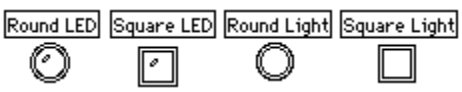


Some Boolean controls that simulate mechanical push-buttons, toggle switches, and slide switches are shown in the following illustration.



Note: The dialog button, dialog checkmark, and dialog radio button look different on each platform. For information about dialog box controls on various platforms, see the Dialog Box Controls section in Front Panel Object Introduction.

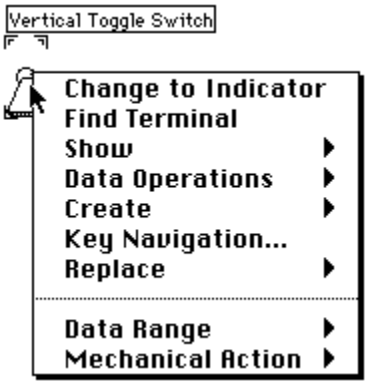
Some Boolean indicators that simulate LEDs and lights are shown in the following illustration.



Clicking on a Boolean control with the Operating tool toggles it between its TRUE (on) and FALSE (off) states. In run mode, clicking on an indicator has no effect because indicators are for output only. In edit mode, you can operate both controls and indicators.

Configuring Boolean Controls and Indicators

Each Boolean control or indicator has several options available in its pop-up menu. The pop-up menu for a Boolean object is shown in the following illustration.



The options above the dotted line in the pop-up menu are common to all controls and indicators and are described in Front Panel Control and Indicator Common Options.

The options below the line in the pop-up menu are described in the following topics:

Labeling Booleans
Stopping on a Boolean Value
Configuring the Mechanical Action of Boolean Controls
Customizing a Boolean with Imported Pictures

Labeling Booleans

Several controls in the **Boolean** palette display text and are called labeled Booleans. Initially, the buttons display the word ON in their TRUE state and the word OFF in their FALSE state, as shown in the following illustration. (When you click on the button with the Operating tool, the control toggles to the opposite state.) In edit mode, you can use the Labeling tool to change the text in either state, for example, YES instead of ON.

By default, the text is centered on the button. If you want to move the text, select **Release Text** from the pop-up menu.

Now you can use the Positioning tool to reposition the text, or choose the option **Lock Text in Center**. Select the Boolean text, then use the **Text** menu to change the font, size, and color of the text. You can also remove the Boolean text from both states by selecting **Hide Boolean Text** from the pop-up menu.

The Boolean controls that do not appear in the palette as labeled Booleans are unlabeled by default. However, you can select **Boolean Text** from the **Show** submenu to make them labeled. You can move the Boolean text in these Booleans. Their text is not locked in the center of the button unless you select **Lock Text in Center** from the pop-up menu.

If you want to change the font of either the name label or the Boolean text without changing both, select what you want to change with the Labeling tool, and then use the **Text** menu options to make the changes you want.

Stopping on a Boolean Value

If you want to recognize when a conditional test fails to produce the correct Boolean value, select **Suspend if True** or **Suspend if False** from the **Data Range** submenu of the Boolean pop-up menu.

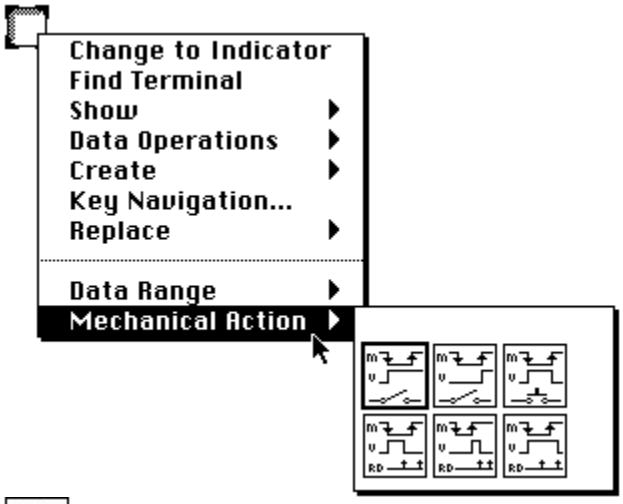
Boolean Range Checking

You may want to detect errors in Boolean values. If you expect a Boolean to always be TRUE, you can select **Suspend If False** from the **Data Range** submenu of the Boolean pop-up menu. If you expect the

Boolean to always be FALSE, you can select **Suspend If True** to catch errors. If the unexpected Boolean value occurs, the VI suspends before or after it executes as described for the **Suspend** option of numeric range.

Configuring the Mechanical Action of Boolean Controls

Boolean controls have six types of mechanical action. Select the appropriate action for your application from the pop-up menu Mechanical Action palette, shown in the following illustration. In these palette symbols, M stands for the motion of the mouse button when you operate the control, V stands for the controls output value, and RD stands for the point in time that the VI reads the control.



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

 The **Switch When Released** action changes the control value only after you release the mouse button during a mouse click within the controls graphical boundary. The action is not affected by how often the VI reads the control.

 The **Switch Until Released** action changes the control value when you click on it, and retains the new value until you release the mouse button. At this time, the control reverts to its original value, similar to the operation of a door buzzer. The action is not affected by how often the VI reads the control.

 With **Latch When Pressed** action, the control changes its value when you click on it, and retains the new value until the VI reads it once. At this point, the control reverts to its default value, whether or not you keep pressing the mouse button. This action is similar to that of a circuit breaker and is useful for stopping While Loops or for getting the VI to do something only once each time you set the control.

 The **Latch When Released** action changes the control value only after you release the mouse button within the controls graphical boundary. When your VI reads it once, the control reverts to the old value. This guarantees at least one new value.

 With **Latch Until Released** action, the control changes value when you click on it, and retains it until your VI reads it once or you release the mouse button, whichever occurs last.

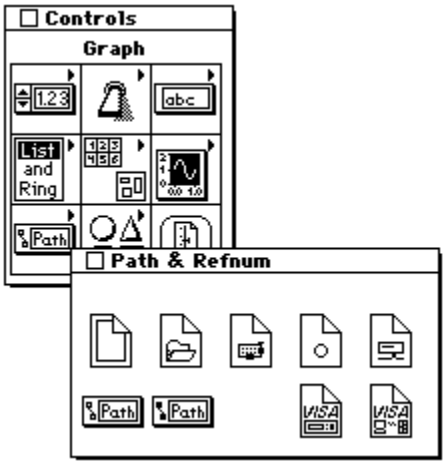
For an example of Boolean controls and indicators, see `examples\general\controls\booleans.llb`.

Customizing a Boolean with Imported Pictures

You can design your own Boolean style by importing pictures for the TRUE and FALSE state of any of the Boolean controls or indicators. This process is explained in Custom Control Creation.

Path Controls and Refnums

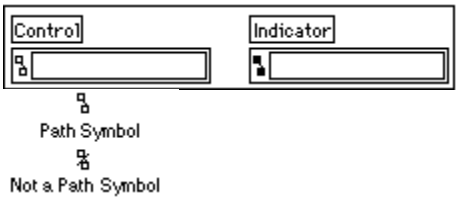
This topic describes how to use file path controls and refnums, which are available from the **Path & Refnum** palette of the **Controls** palette, shown in the following illustration.



Using Path Controls and Indicators
Using Refnums

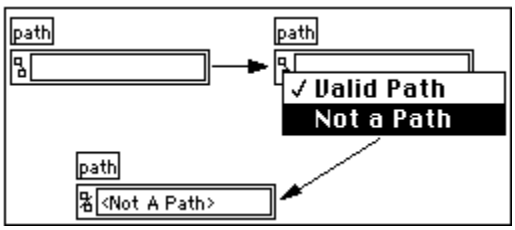
Using Path Controls and Indicators

A path control and path indicator are shown in the following illustration.



The path control and indicator are used to enter and display, respectively, the location of a file or directory in a file system using the standard syntax for a given platform. If a function that is supposed to return a path fails, it returns an invalid path. When an invalid path is displayed by a path control or indicator, the path symbol changes to the Not a Path symbol, and `<Not A Path>` appears in the text field to indicate that the path is invalid.

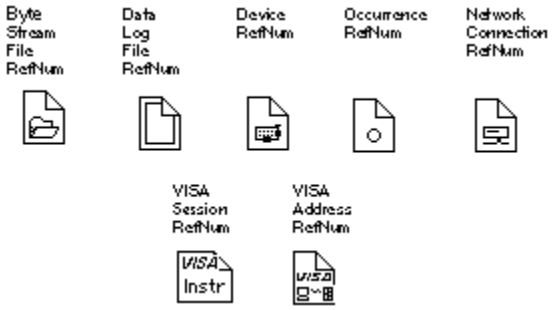
You can change the value of a path control from a valid path to an invalid path by clicking on the path symbol and selecting the **Not a Path** option from the menu. This is shown in the following illustration. In the same way, you can change the value of a path control or indicator from an invalid path to a valid empty path by clicking on the not a path symbol and selecting the **Valid Path** option from the menu.



You might use this **Not a Path** value as the default value for a path control on your VI. This way, you can detect when the user has not entered a path in which case you can give the user a file dialog box for choosing a path.

Using Refnums

The **Path & Refnums** palette from the **Controls** palette contains several refnum controls, shown in the following illustration. These refnums are generally used to identify I/O operations. LabVIEW has seven refnum data types for various kinds of I/O. For each refnum, there is a corresponding control. The following illustration shows the different kinds of refnums.



To open a file, you must specify the path of the file you want to open. Because you can open more than one file at a time in LabVIEW, you need to specify which operations apply to which open file. The path control is not sufficient for this, because you can open one file more than once, concurrently.

When you open a file, LabVIEW returns a *refnum* that identifies that file. You use this refnum in all subsequent operations that relate to that file. You can think of this refnum as a unique number produced each time you open a file to identify it. When you close the file, the refnum is disassociated from it.

Note: You only need to use a refnum control or indicator if you want to pass a refnum between two VIs.

For information on the two VISA refnums, see the LabVIEW Instrument I/O VIs topic.

There are two kinds of file refnums, one for datalog files and one for byte stream files.

Because datalog files have an inherent structure, the Data Log File RefNum is used to pass the refnum as well as a description of the file type to (or from) calling VIs. The Data Log File RefNum is resizable, like a cluster. You place a control inside the refnum that defines the structure of the file. For a file that contains numbers, you create a datalog refnum containing a number. If each record in the file contains a pair of numbers, you place a cluster inside the refnum, and then place two numeric controls inside the cluster.

The remaining refnums are easy to use, because they do not have any options. The Byte Stream File RefNum is used with byte stream files, either text or binary files. It is typically used when you open or create a file in one VI but want to perform I/O on the file in another VI. You need a refnum control on the front panel of the VI that performs I/O, and a refnum indicator on the front panel of the VI that opens or creates the file.

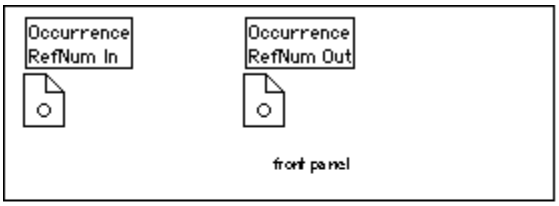
The Device RefNum is used with the device I/O functions available for the Macintosh. It is only used when you open a device in one VI but want to perform I/O on the device in another VI.

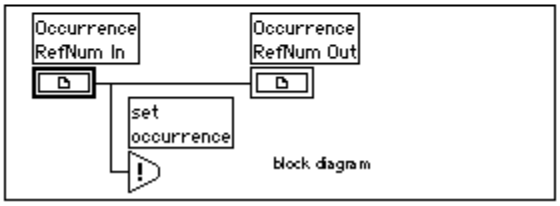
Note: You rarely need to use a Device RefNum control on your front panel. These controls are used in the Instrument I/O VIs in `vi.lib` to access the LabVIEW GPIB and serial device drivers. You only need to use a Device RefNum control if you have to write a special-purpose LabVIEW device driver for the Macintosh.

The Network Connection RefNum is used with the TCP/IP VIs. You would generally use it when you open a network connection in one VI but want to perform I/O on the network connection in another VI.

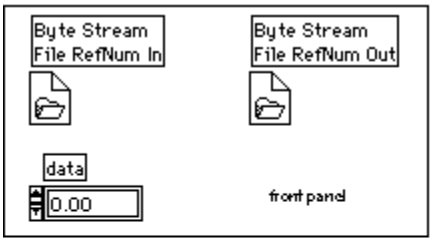
The Occurrence RefNum is used with the occurrence functions. You would use it only when you generate an occurrence in one VI but want to set or wait for the occurrence in another VI.

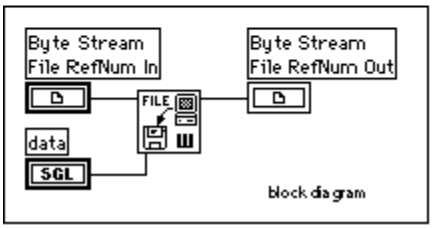
The following illustration shows the front panel and block diagram of a VI that uses and returns an Occurrence RefNum generated in another VI.



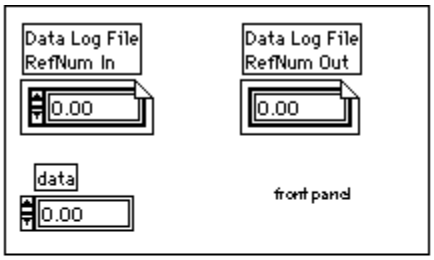


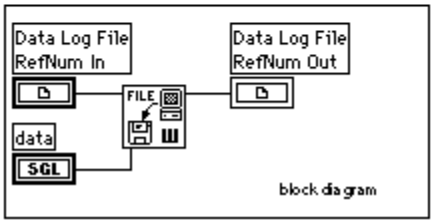
The following illustration shows the front panel and block diagram of a VI that uses and returns a Byte Stream File RefNum generated in another VI.





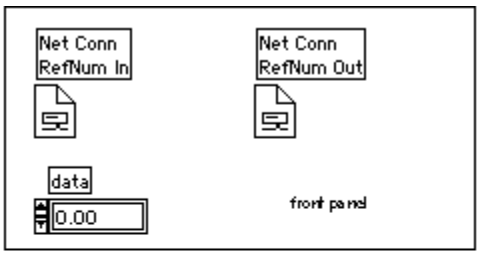
The following illustration shows the front panel and block diagram of a VI that uses and returns a Data Log File RefNum generated in another VI. Datalog files can contain any data type. The datalog file in the following example contains numeric data.

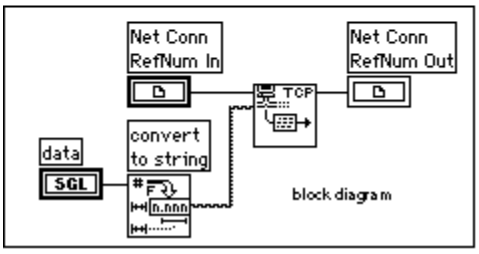




The following illustration shows the front panel and block diagram of a VI that uses and returns a Network

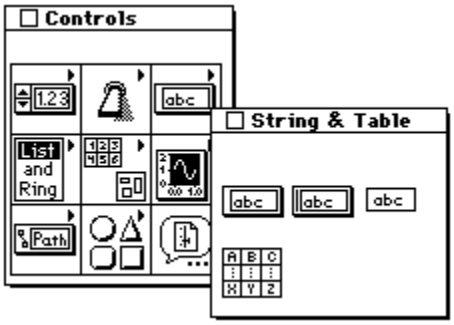
Connection RefNum generated in another VI.





String Controls and Indicators

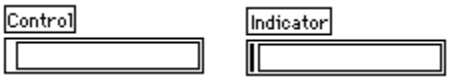
This topic discusses how to use string controls and indicators, and the table. You can access these objects through the **String & Table** palette of the **Controls** palette, shown in the following illustration.



Using String Controls and Indicators
Options for String Controls and Indicators
Using the Table

Using String Controls and Indicators

A string control and string indicator are shown in the following illustration.



You enter or change text in the string control using the Operating tool or the Labeling tool. As with the digital control, new or changed text does not pass to the diagram until you press the <Enter> key on the numeric keypad, click on the enter button in the **Tools** palette, or click outside the control to terminate the edit session. Pressing the <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key on the alphanumeric keyboard enters a carriage return.

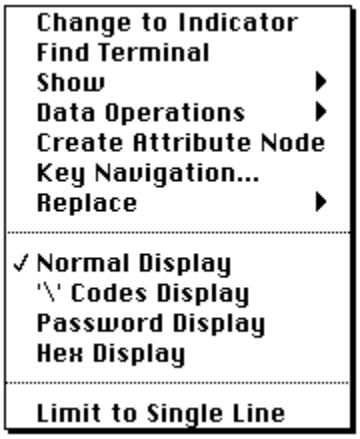
When the text reaches the right border of the display window, the string wraps to the next line, breaking on a natural separator such as a space or tab character.

Note: When running a VI, you use the <Tab> key to move to the next control. When editing a VI, pressing the <Tab> key changes tools. To enter a tab character into a string, select the \ Codes option from the string pop-up menu and type \t. To enter a linefeed into a string, press the <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key on the alphanumeric keyboard, or select the \ Codes option from the string pop-up menu and type \n. To enter a carriage return into a string, select the \ Codes option from the string pop-up menu and type \r. See the table in Backslash (\) Codes Display Option for a complete list of \ codes.

For information on manipulating strings, see the String Functions topic in the Function and VI Reference. Also see the examples in examples\general\strings.llb.

Options for String Controls and Indicators

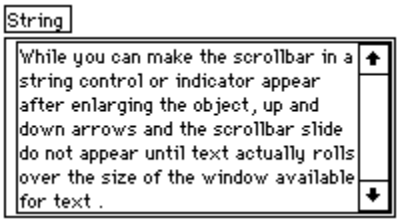
Strings have special features that you can access through the string pop-up menu, shown in the following illustration.



Show Scrollbar Option
Normal Display Option
Backslash (\) Codes Display Option
Password Display Option
Hex Display Option
Limit to Single Line Option

Show Scrollbar Option

The **Show»Scrollbar** option from the string pop-up menu is grayed-out unless you have increased the size of your string enough for a scrollbar to fit. If you select this option, a vertical scrollbar appears on the string control or indicator as shown in the following illustration so that you can display text not visible in the string control. You can use this option to minimize the space taken up on the front panel by string controls that contain a large amount of text. If the menu item is dimmed, you must make the string taller to accommodate the scrollbar before you choose this option.



Normal Display Option

The **Normal Display** option displays all characters as typed (with the exception of characters created with special keys such as the <Tab> or <Esc> key, which are nondisplayable).

Backslash (\) Codes Display Option

Choosing the option **\ Codes Display** from the string pop-up menu instructs LabVIEW to interpret characters immediately following a backslash (\) as a code for nondisplayable characters. The following table shows how LabVIEW interprets these codes.

Code	LabVIEW Interpretation
\00 - \FF	Hex value of an 8-bit character; must be uppercase

\b	Backspace (ASCII BS, equivalent to \08)
\f	Form feed (ASCII FF, equivalent to \0C)
\n	Linefeed (ASCII LF, equivalent to \0A)
\r	Carriage return (ASCII CR, equivalent to \0D)
\t	Tab (ASCII HT, equivalent to \09)
\s	Space (equivalent, \20)
\\	Backslash (ASCII \, equivalent to \5C)

Use uppercase letters for hexadecimal characters and lowercase letters for the special characters, such as formfeed and backspace. Thus, LabVIEW interprets the sequence `\BFare` as hex BF followed by the word *are*, whereas it interprets `\bFare` and `\bfare` as a backspace followed by the words *Fare* and *fare*. In the sequence `\Bfare`, `\B` is not the backspace code, and `\Bf` is not a valid hex code. In a case like this, when a backslash is followed by only part of a valid hex character, LabVIEW assumes a *0* follows the backslash, so LabVIEW interprets `\B` as *hex 0B*. Any time a backslash is not followed by a valid hex character, LabVIEW ignores the backslash character.

You can enter some nondisplayable characters from the keyboard, such as a carriage return, into a string control whether or not you select **\ Codes Display**. However, if you enable the backslash mode when the display window contains text, LabVIEW redraws the display to show the backslash representation of any nondisplayable characters as well as the `\` character itself.

Suppose the mode is disabled and you enter the following string.

```
left
\right\3F
```

When you enable the mode, the following string appears because the carriage return after left and the backslash characters following it are shown in backslash form as `\n\\`.

```
left\n\\right\\3F
```

Suppose now that you select **\ Codes Display** and you enter the following string.

```
left

\right\3F
```

When you disable the mode, the following string appears because LabVIEW originally interpreted \r as a carriage return and now prints one. \3F is the special representation of the question mark (?) and prints this way.

```
left

right?
```

Now if you select **\ Codes Display** again, the following string appears.

```
left\n\right?
```

Indicators behave the same way.

Notice in these examples that the data in the string does not change from one mode to the other. Only the displayed representation of certain characters changes.

The backslash mode is very useful for debugging programs, and for sending non-printable characters to instruments, serial ports, and other devices.

Password Display Option

With the **Password Display** option, the string control displays an * for each character entered into it. When you read the strings data from the block diagram, however, you read the actual data that the user entered. Notice that if you try to copy data from the control, only the * characters are copied.

Hex Display Option

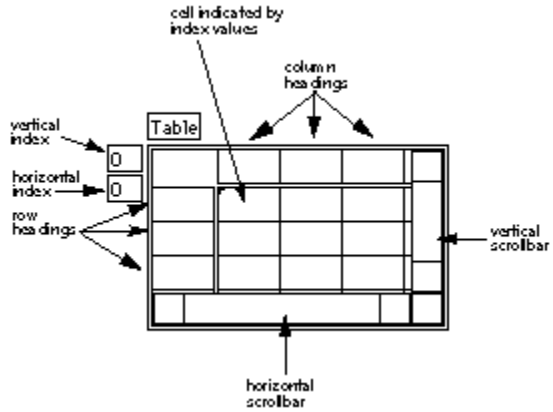
Use the **Hex Display** option to display the string as hex characters rather than alphanumeric characters.

Limit to Single Line Option

The **Limit to Single Line** option prevents you from entering a carriage return into a string while typing into it.

Using the Table

A table is a 2D array of strings. The following illustration is an example of a table with all its features shown.



Resizing a Table, Rows, and Columns
Entering and Selecting Data in a Table

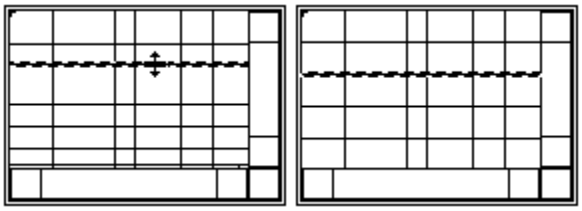
A table has row and column headings which are separated from the data by a thin, open border space. You enter headings when you place the table on the front panel, and you can change them using the Labeling tool or Operating tool. You can update or read headings using the Attribute Node.

The index display indicates which cell is visible at the upper left corner of the table. You can operate these indices just as you do on an array.

Resizing a Table, Rows, and Columns

You can resize the table from any corner with the Resizing tool. You can resize individual rows and columns in a table by dragging one of the border lines with the Positioning tool. When the tool is properly placed to drag a line, one of the drag cursors shown in the following illustration appears. Click and drag to resize the row or column.

You can use the <Shift> key while dragging a border line to size multiple rows or columns to be all the same size. If the row or column you are resizing is inside an area of the table you have selected (outlined in blue or bold), all the rows or columns in the table are also equally sized. Thus the uneven horizontal rows in the following illustration to the left become evenly spaced in the illustration to the right.



Entering and Selecting Data in a Table

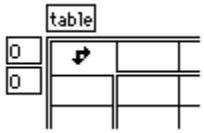
You can use the keyboard to enter data into a table rapidly. Click inside a cell with either the Operating tool or the Labeling tool, and enter your data from the keyboard.

The <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key on the alphanumeric keyboard enters your text and moves the cursor into the cell below. The <Enter> key on the numeric keypad enters your data and terminates the entry. Pressing the <Shift> key while pressing the arrow keys moves the entry cursor through the cells.

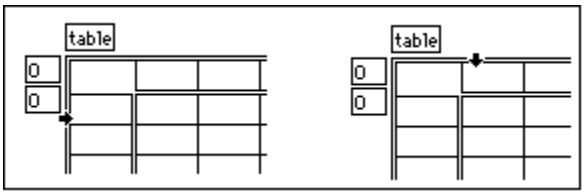
You can select individual cells with the Operating tool or by double-clicking, and you can extend the selection by pressing <Shift-click> and then dragging.

Moving outside the current contents of the table scrolls the table while extending the selection. A border appears around selected cells to indicate that they have been selected. You can turn scrolling on or off using the Selection Scrolling option in the table pop-up menu.

You can also select all the data in a table, or all the data in a row or column. Position the Operating tool at the upper left corner of the table to select all data. A special, double-arrow cursor appears when the tool is positioned properly as shown in the following illustration. Click the mouse to select the data.



To select an entire row or column of data, position the Operating tool at the left border of a row, or at the top of a column. Again, a special arrow cursor appears when the tool is properly positioned, shown in the illustration that follows. Click and drag across width of column or height of row to select the data.



You can copy, cut and paste data, using the **Copy Data**, **Cut Data**, and **Paste Data** options in the **Data**

Operations submenu of the pop-up menu.

If you cut any row or rows of data, all rows below it move up, as shown in the top portion of the following illustration. If you cut any column of data, all columns to the right scroll left, as shown in the bottom portion of the illustration.

Selecting a row | After cutting row

Selecting a column | After cutting column

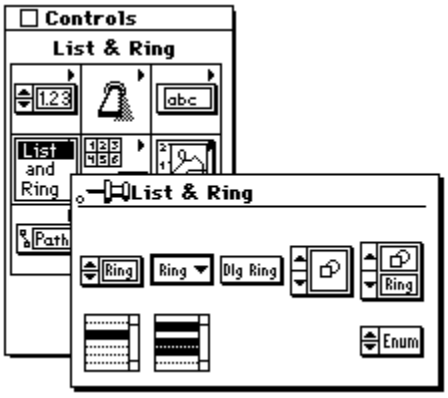
You can turn the option to show a selected area of data on or off by selecting or deselecting **Data Operations»Show Selection**.

The row and column headings of a table are not part of the table data. Table headings are a separate piece of data, and can be read and set using the Attribute Node.

The table is the same as a two-dimensional array of strings as far as the LabVIEW block diagram is concerned. Because of this, LabVIEW string functions can manipulate tables. See the String Functions topic for information on using these functions.

List and Ring Controls and Indicators

This topic describes the LabVIEW list box and ring controls and indicators, which are available from the **List & Ring** palette of the **Controls** palette, shown in the following illustration.



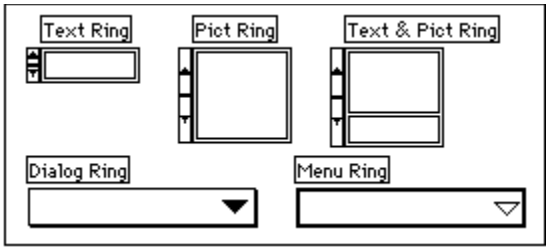
The List & Ring palette contains eight controls--Text Ring, Menu Ring, Dialog Ring, Pict Ring, Text & Pict Ring, Single Selection list box, Multiple Selection list box, and Enumerated Type.

Ring Controls
List Box Controls
Enumerated Type Controls

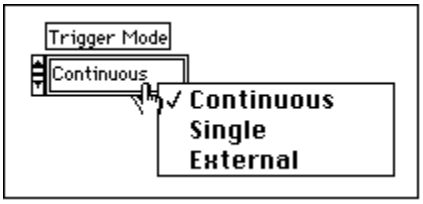
For information about the attributes of list box and ring controls, see Attribute Nodes.

Ring Controls

Rings are special numeric objects that associate unsigned 16-bit integers with strings, pictures, or both. The different styles of rings are shown in the following illustration.



Rings are particularly useful for selecting mutually exclusive options, such as trigger modes. For example, you may want users to be able to choose from continuous, single, and external triggering, as shown in the following illustration.



In the preceding example, the ring labeled `Trigger Mode` has three mode descriptors, one after another, in the text display of the ring. LabVIEW arranges items in a circular list (like a Rolodex) with only one item visible at a time. Each item has a numeric value, which ranges from 0 to *n*-1, where *n* is the number of items (3 in this example). The value of the ring, however, can be any value in its numeric data

range. It displays the last item (`External`, above) for any value ³ 2 and the first item (`Continuous`, above) for any value # 0.

Users can use this ring to choose an option from an easy-to-understand list without having to know what value that option represents. The value associated with the selected option passes to the block diagram, where, for example, you can use it to select a case from a Case Structure (conditional code) that carries out the selected option.

You can select an item in a ring control two ways. You can use the increment buttons to move to the next or previous item. Incrementing continues in circular fashion through the list of items as long as you hold down the mouse button. You can also select any item directly by clicking on the ring with the Operating tool and then choosing the item you want from the menu that appears.

Adding Text Items to a Ring
Adding Picture Items to a Ring
Changing the Size and Text of a Text & Pict Ring

Adding Text Items to a Ring

A new text ring has one item with a value of zero and a display containing an empty string. You enter or change text in the ring text area as you do with labels, using the Labeling tool. Press the `<Enter>` (Windows and HP-UX), or `<Return>` (Macintosh and Sun) key or click outside the text area to finish typing. **Add Item After** from the text ring pop-up menu creates a new empty item following the current one. **Add Item Before** inserts a new item in front of the current item. If you are editing item 0 when you select **Add Item After**, for example, LabVIEW creates item 1, ready for you to enter text for the new item. You can also press `<Shift-Enter>` (Windows and HP-UX), or `<Shift-Return>` (Macintosh and Sun) after typing in an item to advance to a new item.

You add items to the menu ring or dialog ring the same way you add items to the text ring. However, the menu ring looks and operates like a pull-down menu. It has no increment buttons. You select an item by clicking on the ring and selecting an item from the menu that appears. The ring displays the currently selected item. You can pop up and select **Show»Digital Display** to display the numeric value associated with the current item of a ring.

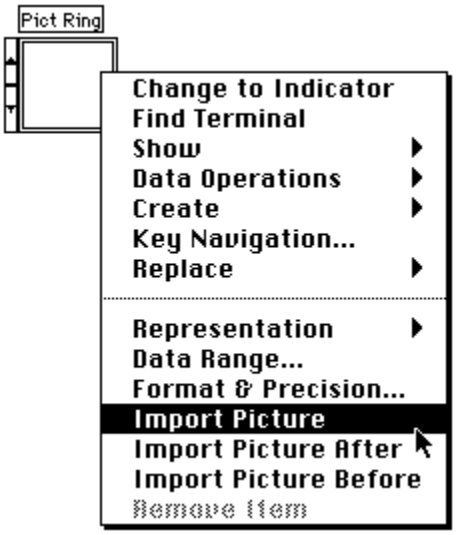
For every ring, the item values above the insertion point increase by one to adjust for the new item. For example, if you insert an item after item 4, the new item becomes item 5, the previous item 5 becomes item 6, 6 becomes 7, and so on. If you insert an item before item 4, the new item becomes item 4, the previous item 4 becomes item 5, 5 becomes 6, and so on.

Use the **Remove Item** command from the ring pop-up menu to delete any item. As with the add item commands, the numeric values of the item adjust automatically.

If you set a ring indicator or control to a value smaller than zero or greater than the number of items included, the control or indicator displays the first or the last item, respectively.

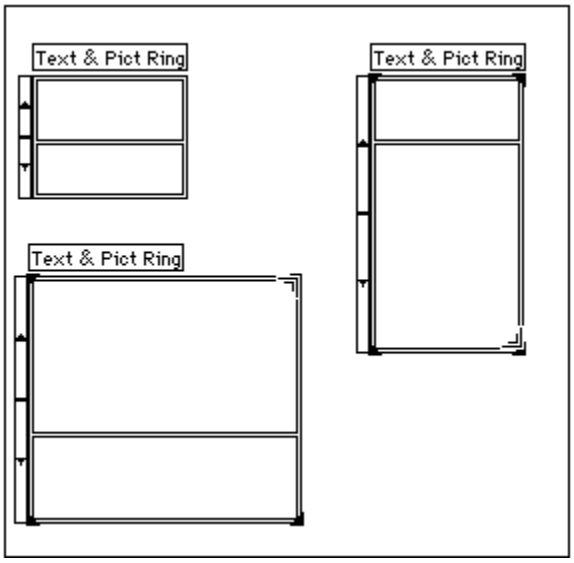
Adding Picture Items to a Ring

A new Pict Ring or Text & Pict Ring has one item with an empty picture display. You must copy a picture to the LabVIEW Clipboard before you can import the picture into a Pict Ring. When you have a picture on the Clipboard, pop up on the Pict Ring and select **Import Picture**. To add another picture, copy it to the Clipboard, then pop up on the Pict Ring and select **Import Picture Before** or **Import Picture After**, as shown in the following illustration.



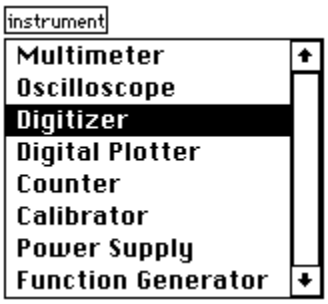
Changing the Size and Text of a Text & Pict Ring

You can enlarge the Text & Pict Ring to make more room for the contents, just as you can all controls. You can also decrease the size. When you resize the Text & Pict Ring from one of its bottom corners, the height of the text area changes. When you resize the Text & Pict Ring from one of its top corners, the height of the picture area changes. These resizing capabilities are shown in the following illustration.



List Box Controls

List box controls are used to present users with a list of options. An example is shown in the following illustration. There are two types of list box controls--Single Selection in which only one option can be selected, and Multiple Selection, in which one or more options can be selected.



In addition to displaying a list of text options, you can have one of several different symbols drawn next to each item (as in the Save dialog box, where directories and files have different symbols). You can detect the currently selected option(s) by reading the controls value. You can also detect on which item, if any, the user double-clicked by using an Attribute Node. Finally, you can choose to have some options grayed out (non-selectable) using the Attribute Node.

Creating the List of Options
Selecting an Item
Data Type of List Box Controls
List Box Pop-up Menu
Creating a Grayed-Out Dividing Line

Creating the List of Options

When you put a list box on a front panel, it contains no text. You can create the list of options in edit mode by typing them with the Labeling tool, or in run mode by using the Attribute Node. Separate individual options by entering a carriage return. If you can determine the options when you create the VI, you may want to type in the options while in edit mode. If you can only determine the options at run time (for example, a list of files), you should use the Attribute Node.

Selecting an Item

There are three ways to select items in a list box:

Using the Mouse
Using the Arrow Keys
Typing Part of the Name of the Desired Option

Using the Mouse

With both the Single Selection and Multiple Selection list boxes, you can select (highlight) an item by clicking on it with the Operating tool. If you click on a different item, then the first item is deselected, and the new item is selected.

In addition, with the Multiple Selection list box, you can select more than one item by `<Shift>`-clicking on additional items.

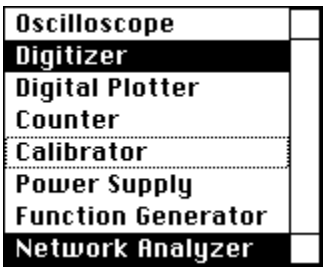
Using the Arrow Keys

If you make the list box the active control (by tabbing to it, clicking on it, or setting key focus using the Attribute Node), you can use the arrow keys to select options within the control.

With the Single Selection list box, you can use the arrow keys to select the next or previous selectable item.

With the Multiple Selection list box, you can also use the arrow keys. As you press the arrow keys, an

outline box moves from item to item, as shown in the following illustration.



To add the currently outlined option to the list of selected items, press the spacebar. You can deselect a currently selected option by using the arrow keys to move to the option and pressing the spacebar. In the preceding illustration, two options (**Digitizer** and **Network Analyzer**) are selected, while a third item (**Calibrator**) has been outlined. The third item can be added to the list of selections by pressing the spacebar.

Typing Part of the Name of the Desired Option

If you make the list box the active control (by tabbing to it, clicking on it, or setting key focus using the Attribute Node), you can type part of an options name to select an option within the control.

With the Single Selection list box, you can select an option by typing a portion of the name of the option. For example, in the previous example, if you type the letter *C*, the selection moves to the **Counter** option. By quickly typing additional letters, you can clarify which option you really want to select. Thus, if you next type an *a*, the **Calibrator** option is selected. If you type ○, the **Counter** option remains selected. If you make a mistake, wait a few seconds and start over with the first letter.

You can use the left and right arrow keys to go to previous or next items that match the typed letters.

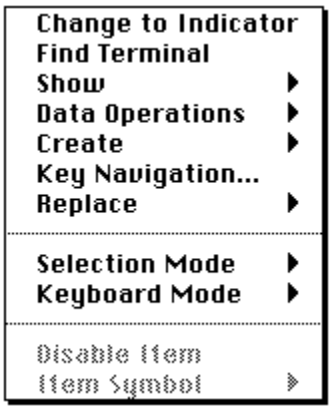
Data Type of List Box Controls

The data type for a Single Selection list box is Int32. The data value for the Single Selection list box is a number that represents the currently selected option, with the first option having a value of zero. If no option is selected, the value is -1.

The Multiple Selection list box is an array of Int32s, where the value(s) in the array represent the currently selected option(s). If no option is selected, the value is an empty array.

List Box Pop-up Menu

The list box pop-up menu includes the options shown in the following illustration. Only the options unique to list box controls are described in this topic.

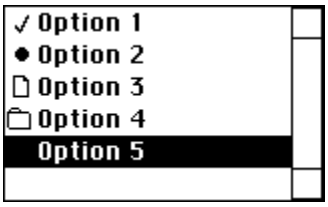


Show Option
Selection Mode
Keyboard Mode

Show Option

You use **Show** from the pop-up menu, shown in the following illustration, to select which components of the list box are visible. You can optionally show or hide the label, the scrollbar, and the symbols.

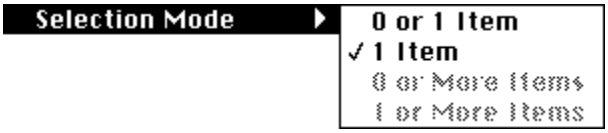
If you select **Show»Symbols**, the list box adds an extra column to the left side of the list for symbols. An example of this is shown in the following illustration. Initially, no items have symbols. You can add symbols using the Attribute Node. See Attributes for List Box Controls for information on supported symbols.



Selection Mode

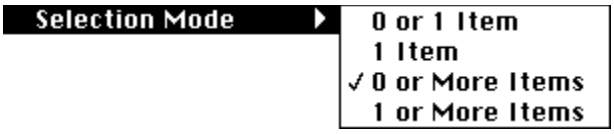
You use **Selection Mode** from the pop-up menu to specify how many items you can select at once.

For a Single Selection list box, you have selection mode options to set the list box to support zero or one selections, or one selection, shown in the following illustration.



If you select **1 Item** (the default setting), then one option in the list box is always selected (highlighted). Whenever you click on a different option, the new option is selected (highlighted), and the previous option is deselected. If you select **0 or 1 Item**, the behavior is the same, with the exception that a user can deselect the current option by `<Shift>`-clicking on it, leaving no options selected.

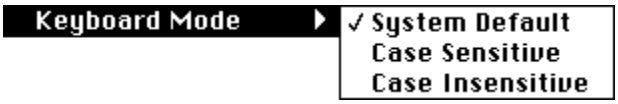
In a Multiple Selection list box, you have the same selection mode options as the Single Selection list box. In addition, you can specify that multiple options can be selected, as shown in the following illustration.



By default, the Multiple Selection list box has a selection mode that allows the user to **select 0 or More Items**. You can also select **0 or More Items** to put it into a multiple selection mode where at least one item must be selected.

Keyboard Mode

You use **Keyboard Mode** from the pop-up menu, shown in the illustration that follows, to specify how the list box should treat upper and lower case characters when you select options by typing their initial letters.



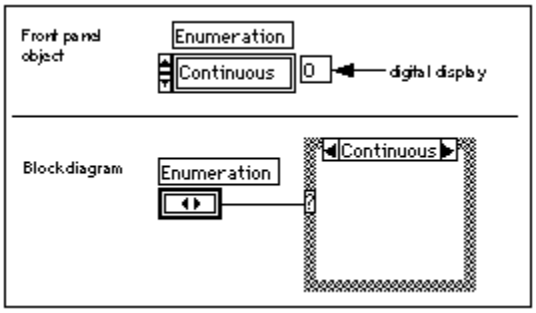
If you select **Case Sensitive**, an item is selected only if the upper- and lower-case letters of what the user types exactly match the item. If you select **Case Insensitive**, case is ignored. If you select **System Default** (the default setting for a list box), the list box behaves the same way that other list boxes do for a given platform. Specifically, on the Macintosh and in Windows, the list box is case insensitive. On the Sun and HP-UX, the list box is case sensitive.

Creating a Grayed-Out Dividing Line

If you use a value of -1 for the symbol of an option, the list box draws a grayed-out dividing line instead of drawing the name of the option.

Enumerated Type Controls

An enumerated type control is similar to a text ring control. If data from an enumerated type is connected to a Case Structure, however, the case displays the string, or text of the item, instead of a number. The enumerated type data type is unsigned byte, unsigned word, or unsigned long, selectable from the **Representation** palette. An example is shown in the following illustration.



One advantage of enumerated types over ring controls is that if you use an enumerated type on a subVIs connector pane, then popping up to create a constant, control, or indicator will create an enumeration with the proper strings.

You enter items into an enumerated type the same way you enter items into a ring, using **Add Item After** or **Add Item Before**. Or you can use the Labeling tool. Press <Shift-Enter> (Windows and HP-UX), or <Shift-Return> (Macintosh and Sun) to enter a new item. Click outside the enumerated type when you have finished entering items.

If an enumerated type is wired to a Case Structure, the Case Structure must have one frame for each item in the enumerated type. Pop up on the Case Structure and select **Add Case After** if you need more

frames. Like a ring, the value of an enumerated type is the index of its current items. You can pop up and select **Show»Digital Display**.

All arithmetic operations except Add One and Subtract One treat the enumerated type the same as an unsigned numeric. Add One increments the last enumeration to the first, and Subtract One decrements the first enumeration to the last.

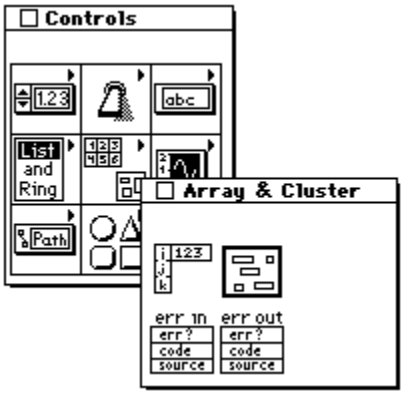
If you connect a numeric value to an enumerated type indicator, LabVIEW converts the number to the closest enumeration item. If you connect an enumeration control to a numeric value, the value is the enumerated type index. To wire an enumeration control to an enumeration indicator, the enumeration items must match exactly.

Arrays

This topic describes how to use LabVIEW arrays, which are available from the **Array & Cluster** palette of the **Controls** palette.

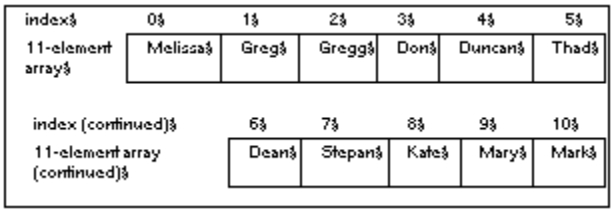
The Function and VI Reference describes LabVIEW functions, many of which can operate on arrays in addition to scalars. The Array Functions topic describes the functions designed exclusively for array operations.

You access arrays from the **Array & Cluster** palette of the **Controls** palette, shown in the following illustration.



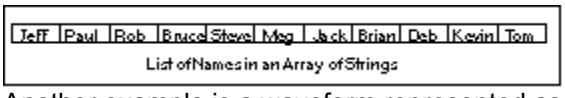
See the examples in `examples\general\arrays.llb`.

An array is a variable-sized collection of data elements that are all the same type, as opposed to a cluster, which is a fixed-sized collection of data elements of mixed types. Array elements are ordered, and you access an individual array element by *indexing* the array. The *index* is zero-based, which means it is in the range of zero to *n*-1, where *n* is the number of elements in the array, as in the following illustration.

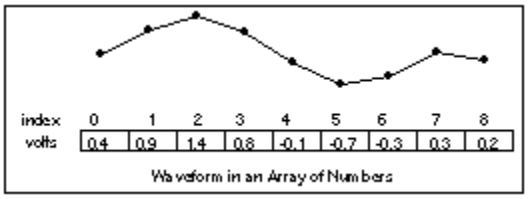


Creating Array Controls
Operating Arrays
LabVIEW Arrays and Arrays in Other Systems

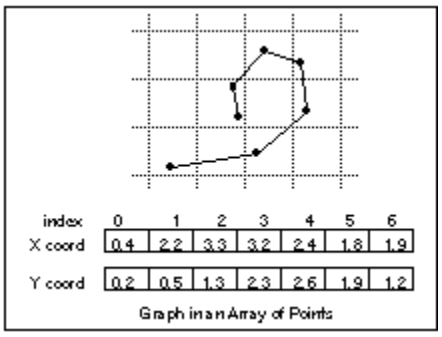
A simple example of an array is a list of names, represented in LabVIEW as an array of strings, as shown in the following illustration.



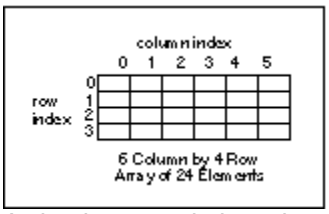
Another example is a waveform represented as a numeric array in which each successive element is the voltage value at successive time intervals, as shown in the following illustration.



A more complex example is a graph represented as an array of points where each point is a pair of numbers giving the X and Y coordinates, as shown in the following illustration.

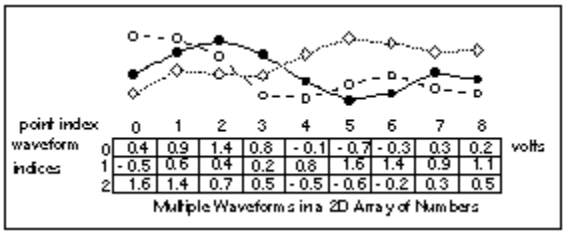


The preceding examples are *one-dimensional* arrays. A *two-dimensional* array requires two indices to locate an element--a column index and a row index, both of which are zero-based. In this case we speak of an N column by M row array, which contains N times M elements, as shown in the following illustration.



A simple example is a chess board. There are eight columns and eight rows for a total of 64 positions, each of which can be empty or have one chess piece. You could represent a chess board in LabVIEW as a two-dimensional array of strings. Each string would have the name of the piece occupying the corresponding location on the board, or it would be an empty string if the location was empty. Other familiar examples include calendars, train schedule tables, and even television pictures, which can be represented as two-dimensional arrays of numbers giving the light intensity at each point. Familiar to computer users are spreadsheet programs, which have rows and columns of numbers, formulas, and text.

All of the one-dimensional array examples can be generalized to two dimensions. A collection of waveforms represented as a two-dimensional array of numbers is shown in the following illustration. The row index selects the waveform and the column index selects the point on the waveform.

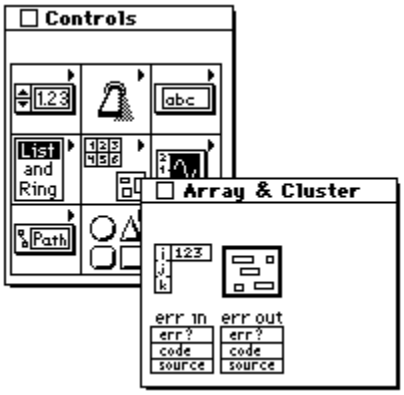


Arrays can have an arbitrary number of dimensions, but one index per dimension is required to locate an element. The data type of the array elements can be a cluster containing an assortment of types, including arrays whose element type is a cluster, and so on. You cannot have an array of arrays. Instead, use a multidimensional array or an array of clusters of arrays.

You should consider using arrays whenever you need to work with a collection of similar data. Arrays are frequently helpful when you need to perform repetitive computations or I/O. Using arrays makes your application smaller, faster, and easier to develop because of the large number of array functions and VIs in LabVIEW.

Creating Array Controls

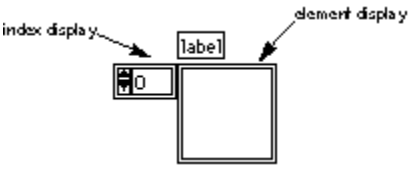
You create an array control in LabVIEW by first selecting an array from the **Array & Cluster** palette of the **Controls** palette, shown in the following illustration.



Setting the Array Dimension
Interpreting the Array Index Display
Displaying an Array in Single-Element or Tabular Form

You create a LabVIEW array control or indicator by combining an *array shell* from the **Array & Cluster** palette of the **Controls** palette, as shown in the preceding illustration, with a valid *element*, which can be a numeric, Boolean, string, or cluster. The element cannot be another array or a chart. Also, if the element is a graph then only the graph data types that contain clusters instead of arrays at the top level are valid.

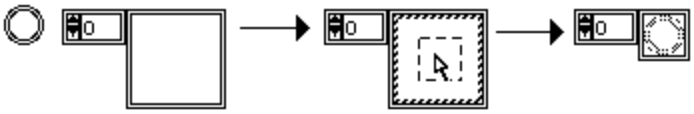
Selecting an array from the **Array & Cluster** palette places an array shell on the front panel, as shown in the following illustration.



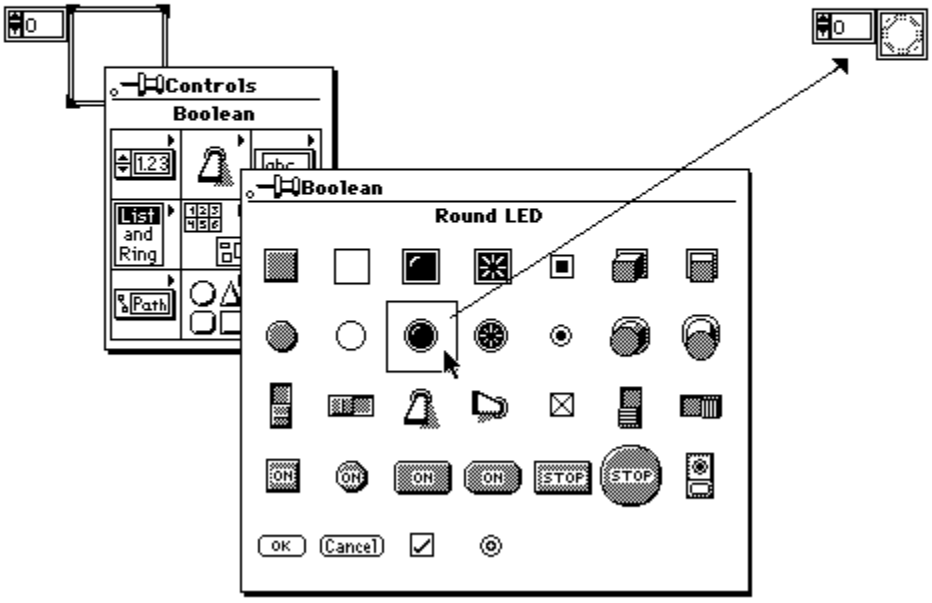
A new array shell has one *index display*, an empty *element display*, and an optional label.

There are two ways to define an array type. You can drag a valid control or indicator into the element display window, or you can deposit the control or indicator directly by popping up in the element display area of the array and selecting a control. Either way, the control or indicator you insert fills the empty display.

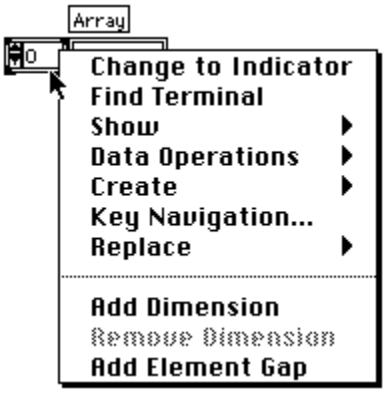
For example, if you want to define an array of Booleans, you can either drag a Boolean into the element display area, or you can pop up in the empty element area and select a Boolean control. Both methods are shown in the following illustrations.



or

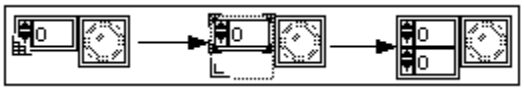


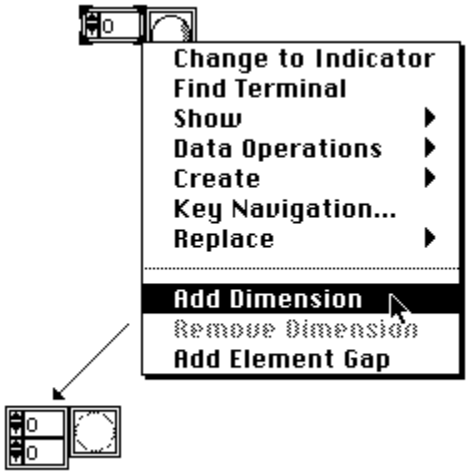
The pop-up menu of the index display is shown in the following illustration.



Setting the Array Dimension

A new array has one dimension and one index display. You resize the index vertically or select the **Add Dimension** command from the index display pop-up menu to add a dimension. You shrink the index vertically or select **Remove Dimension** to delete it. An additional index display appears for each dimension you add. Two methods for resizing are shown in the following illustrations.





The dimension of an array is a choice you make when you decide how many identifiers it takes to locate an item of data. For example, to locate a word in a book you might need to go to the sixth word on the twenty-eighth line on page 192. You need three indices, 6, 28, and 192 to specify the word, so one possible representation for this book is a three-dimensional array of words. To locate a book in a library you would specify the position on the shelf, which shelf, which bookcase, which aisle, and which floor, so if you represented this as an array it would use five dimensions. To specify a word in a library would then require eight indices.

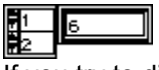
Interpreting the Array Index Display

If you think of a two-dimensional array as rows of columns, the top display is the row index and the bottom display is the column index. The combined display at the right shows the value at the specified position, as shown in the illustration that follows.

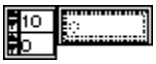
For example, say you have an array of 3 rows and 4 columns, with values as shown in the following table.

0	1	2	3
4	5	6	7
8	9	10	11

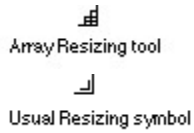
Rows and columns are zero-based, meaning that the first column is column 0, the second column is column 1, and so forth. Changing the index display to row 1, column 2 displays a value of 6, as shown in the following illustration.



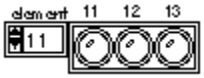
If you try to display something out of range, the value display is grayed out to indicate there is no value currently defined for that value. For example, if you tried to display an element in row 10 of the array shown previously, the display would be grayed out as shown in the following illustration.



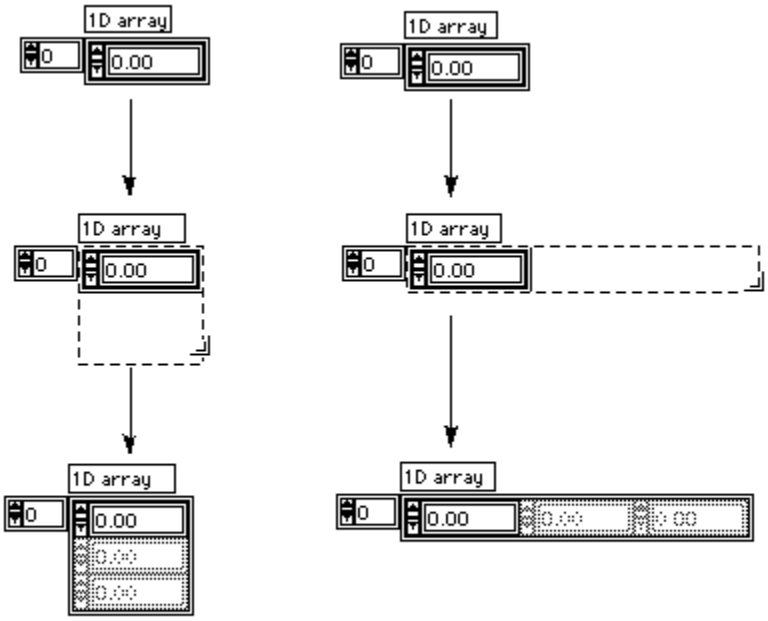
Displaying an Array in Single-Element or Tabular Form



A new array appears in single-element form. The array displays the value of a single element--the one referenced by the index display. You can also display the array as a table of elements by resizing the array shell from any of the four corners surrounding the element. The array Resizing tool is slightly different from the usual Resizing tool when it is over the array shell resizing handles, and the cursor changes to the usual symbol when you begin to resize. An example is shown in the following illustration.

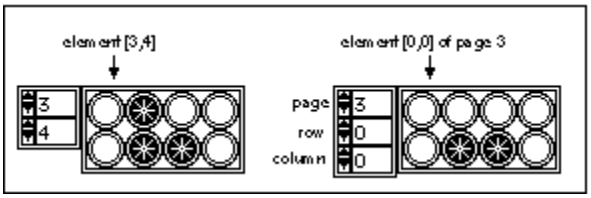


The following illustration shows how you could resize a one-dimensional array either vertically or horizontally to show more elements simultaneously.

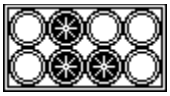


The index display contains the index of the element in the left or upper left corner of the table. By changing the index, you can display different sections of a large or multidimensional array. For one-dimensional (1D) arrays, the index identifies the column of the left-most visible element. For two-dimensional (2D) or higher arrays, the bottom two indices identify the coordinates of the upper left visible element. In the left portion of the following illustration, this element is [3,4].

If you view a three-dimensional (3D) array as a book of pages that are composed of lines (rows) of characters (columns), the table displays part or all of one page. The figure at the right in the following illustration shows the first four columns of the first two rows of page 3 of the array.



If you want to hide the array indices, pop up on the outer frame and turn off the **Show»Index Display** option from the array pop-up menu. The following illustration shows an example of what a tabular array looks like without the index display.



If you want to display an array in an orientation different from the tabular array format, you can display the elements in a cluster on the front panel and process them in an array. To do this, use the Array To Cluster and Cluster To Array functions.

Operating Arrays

You operate the arrays element exactly as you would if it were not part of the array. You operate the index display the same way you operate a digital control. To set the value of an array control element, bring the element into view with the index display, then set the elements value.

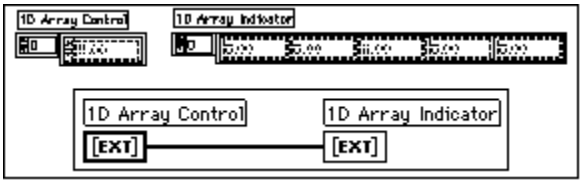
Setting the Default Size and Values of an Array
Selecting Array Elements
Finding the Size of an Array
Moving or Resizing the Array
Selecting Array Cells

Setting the Default Size and Values of an Array

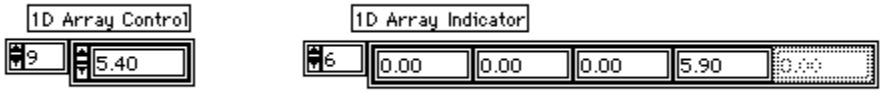
You cannot fix the size of a LabVIEW array. However, when you set the default values of the array, you also set the default size. *Do not make the default size larger than necessary*. Keep in mind that if you set an array to have a large default value, all of the default data is saved along with the VI, thus increasing the size of your file.

A new array shell without an element is *undefined*. It has no data type and no elements and it cannot be used in a program. After you give the array an element, the array is an *empty array* (its length is 0). Although it also has no defined elements, you can still use it in the VI. LabVIEW dims all elements in the array, indicating they are undefined.

The following illustrations show an array control in single-element form that is wired to an array indicator in tabular form with five elements displayed. The first figure shows that both arrays are empty (element values are undefined).

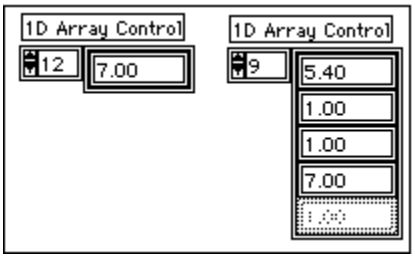


If you set the index display of the array control to 9 and set the element value at index 9 to 5.4, the data in the array expands to 10 elements (0 to 9). The value of element 9 is the value you set, 5.4 in this example. LabVIEW gives the other elements the default value of the digital control, 0.00. After running the VI, the indicator displays the same values.



Selecting **Reinitialize to Default** for an element resets that element only to its default value.

Now assume you change the default value of a numeric element from 0.0 to 1.0. You can do this by changing the value in the indicator with the Operating tool and then popping up on the element and selecting **Make Current Value Default**. Or you can choose the **Data Range...** option from the pop-up menu. Assume you also set the value of element 12 to 7.0. The array now has 13 elements. The values of the first 10 elements do not change, while elements 10 and 11 have the new default value of 1.0, and element 12 has the assigned value of 7.0. This example is shown in the following illustration.



If you now select **Reinitialize to Default** from either the shell or the index pop-up menus, the array reverts to the 10-element array shown in the previous example.

To increase the size of an array from [Ni by Nj by Nk...] to [Mi by Mj by Mk...], assign a value to the new last element [Mi-1, Mj-1, Mk-1...]. To decrease the size, first execute the **Data Operations»Empty Array** command from the index array pop-up menu, then set the array and values to the size you want, or select the unwanted subset of the array and select the **Data Operations»Cut Data** option from the pop-up menu.

Selecting Array Elements

When in run mode, you can use the <Tab> key either to move the key focus between front panel controls or between elements within a single array. Initially, the <Tab> key moves the key focus between front panel controls. To have it move between the elements within a specific array, first <Tab> to that array. Then use the <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key and the down arrow, which allows you to move between the array elements. If you want to return to tabbing between controls, use the <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key and the up arrow.

Finding the Size of an Array

To find the size of an array control or indicator, select **Data Operations»Show Last Element** from the index array pop-up menu.

Moving or Resizing the Array

Always move the array by clicking on the shell border or the index display and then dragging. If you drag the element in the array, it separates from the array, because the shell and elements are not locked together. If you inadvertently drag the element when you meant to drag the shell, cancel the operation by dragging the element back into the shell or past the front panel window border before releasing the mouse button. You also cancel a resizing operation by dragging the border past the front panel window before releasing the mouse button. If the array is part of the current selection, then grabbing an element drags the array.

You can resize the array index vertically from any corner, or horizontally from the left.

Selecting Array Cells

You can copy data from and paste data to LabVIEW arrays by taking these steps.

1. To select the data, set the array index to the first element in the data set you want to copy.
2. Then, select the **Data Operations»Start Selection** option from the array pop-up menu.
3. Next, set the array index to the last element in the data set you want to copy.
4. Finally, select the **Data Operations»End Selection** option from the array pop-up menu.

When you are in run mode, these options are directly available when you pop up on an array. The selection process is detailed in the following example.

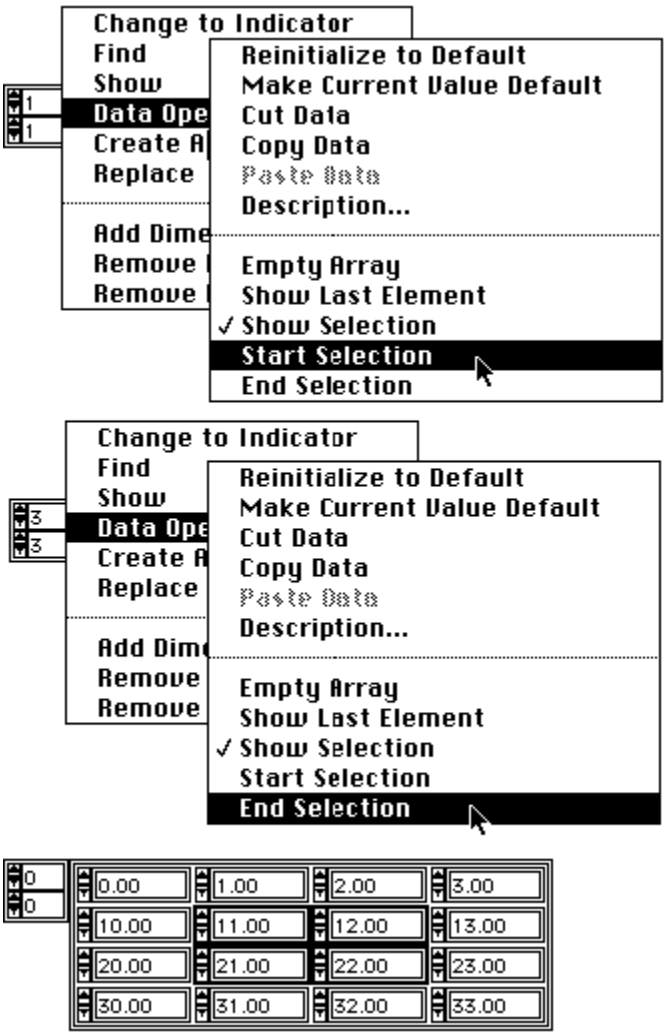
Example of Selecting Array Cells

Begin by selecting **Show Selection** from the pop-up menu. **Show Selection** must be activated for your selected cells to be visibly marked.

A border surrounds selected cells. You can select **Add Element Gap** from the pop-up menu. This separates the cells by a thin open space. While it is not necessary to add the element gap, this open border improves the appearance of the array. When you select cells, this open space is filled by a thick border, denoting the selected cells. If you do not add the element gap, the selection border appears over the selected cell edge.

You define the cells for selection with the array index. The following example uses a two dimensional array.

Set the array index for 1,1. Choose **Start Selection** from the pop-up menu. Set the array index to 3,3. Choose **End Selection** from the pop-up menu. The selected cells are surrounded by a blue border (thick black on a monochrome system).

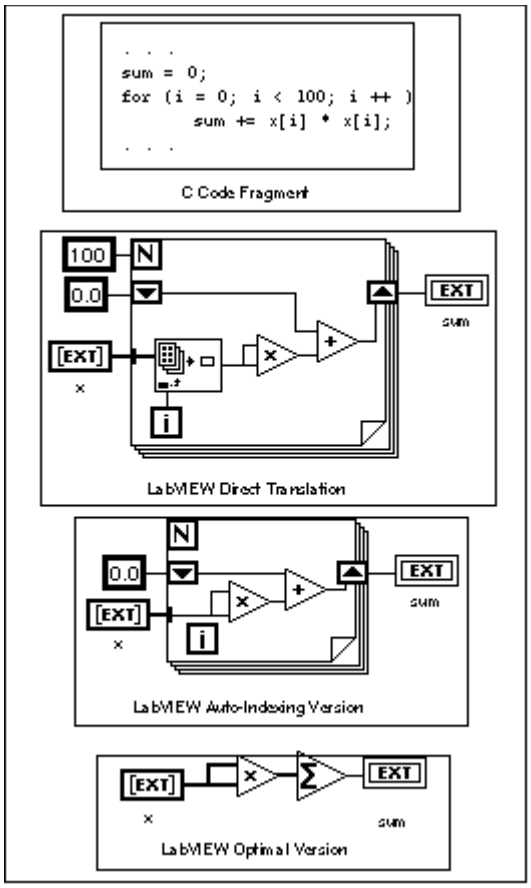


Notice that the selection includes the cells denoted by lower index numbers but not the cells denoted by the higher index numbers. If you want to include the cells from 1,1 to 3,3, you would have set the index to 4,4 before choosing **End Selection**.

After you select cells, you can cut or copy the data to paste into other cells. When you finish, you will notice that one border line remains highlighted. This is an insertion point, and remains in the array. To hide this line, you can deselect the **Show Selection** option from the pop-up menu. You can eliminate this line without deselecting **Show Selection** by emptying the selection. Set the index to 0 in all dimensions, and then select **Start Selection** and **End Selection**. This makes the selection run from 0 to 0 in all dimensions, which is what the selection is set to when you first place it on the front panel.

LabVIEW Arrays and Arrays in Other Systems

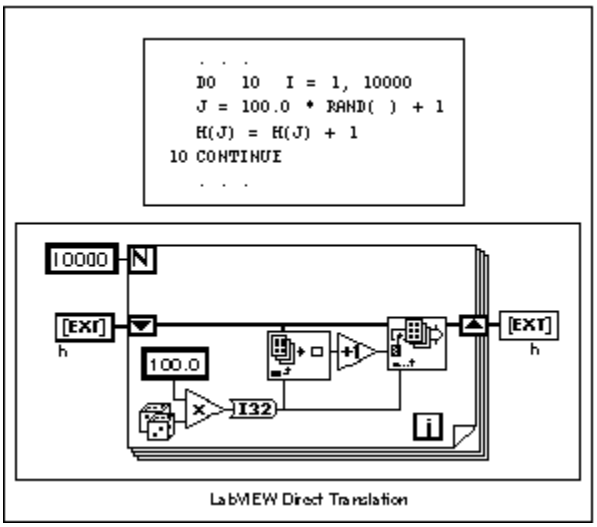
Most programming languages have arrays, although few have the number of built-in array functions that LabVIEW has. Other languages typically stop at the fundamental array operations of extracting an element or replacing an element, leaving it to the users to build more complex operations themselves. LabVIEW has these fundamental operators, so you can directly map a program in another language to a LabVIEW VI, but you can usually create a simpler and smaller diagram if you start over in LabVIEW and use LabVIEWs higher-level array functions. The example that follows shows a fragment of a C program that sums the squares of the elements of an array, the direct translation to a LabVIEW diagram, and two redesigned versions that are more efficient than the translation.



Index Array
function icon

The indexing operation for extracting an element is represented in C by brackets [ ] following the array name and enclosing the index. In LabVIEW, this operation is represented by the Index Array function icon shown at left. The array is wired to the top left terminal, the index is wired to the bottom left terminal, and the array element value is wired to the right terminal.

The following example shows the fundamental operation of replacing an array element. The fragment of FORTRAN code generates a histogram of random numbers (the random number generator produces values between 0 and 1), and so does the LabVIEW version. You can disregard the +1 in the FORTRAN computation of J; the addition is needed because FORTRAN arrays are one-based.



Replace Array Element function icon

The indexing operation for inserting or replacing an element is represented in FORTRAN by parentheses( ) on the left side of the equal sign following the array name, and enclosing the index. In LabVIEW this operation is represented by the Replace Array Element function icon (shown at left). The array is wired to the top left terminal, the replacement value is wired to the middle left terminal, the index is wired to the bottom left terminal, and the updated array is wired from the right terminal.

Before you can use an array in most programming languages you must first declare it and in some cases initialize it. In LabVIEW, building an array control or indicator on the front panel is equivalent to declaring an array. Defining a default value for it is equivalent to giving the array an initial value.

Most languages require you to specify a maximum size for an array as part of the declaration. The information about the size of the array is not treated as an integral part of the array, which means you have to keep track of the size yourself. In LabVIEW, you do not have to declare a maximum size for arrays, because LabVIEW automatically remembers the size information for the array. Furthermore, with LabVIEW, you cannot store a value outside the bounds of an array. In many conventional programming languages there is no such checking, and consequently you can inadvertently corrupt memory.

Clusters

This topic describes how to use LabVIEW clusters, which are available from the **Array & Cluster** palette of the **Controls** palette.

The Cluster Functions topic describes the functions designed exclusively for cluster operations.

You access clusters from the **Array & Cluster** palette of the **Controls** palette, shown in the following illustration.

See the examples in `examples\general\arrays.llb`.

A LabVIEW cluster is an ordered collection of one or more elements, similar to structures in C and other languages. Unlike arrays, clusters can contain any combination of LabVIEW data types. Although cluster and array elements are both ordered, you access cluster elements by *unbundling* all the elements at once rather than indexing one element at a time. Clusters are also different from arrays in that they are of a fixed size. As with an array, a cluster is either a control or an indicator; a cluster cannot contain a mixture of controls and indicators.

Creating Clusters
Operating and Configuring Cluster Elements
Assembling Clusters
Disassembling Cluster Elements
Replacing Cluster Elements

You can use clusters to group related data elements that appear in multiple places on the diagram, which reduces wire clutter and the number of connector terminals subVIs need.

Most clusters on the block diagram have a common wire pattern, although clusters of numbers (sometimes referred to as *points*) have a special wire pattern. These wiring patterns are shown in the following illustration.

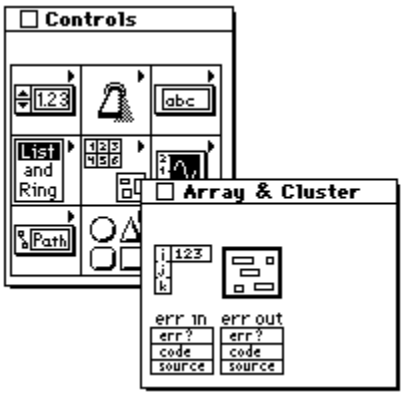
Common Cluster Wire Pattern
Cluster of Numbers Wire Pattern

You can connect terminals only if they have the same type. For clusters, this means that both clusters must have the same number of elements, and corresponding elements--determined by the cluster order--must match in type. LabVIEW coerces numbers of different representations to the same type.

For information on functions that manipulate clusters, see the Cluster Functions topic.

Creating Clusters

You create a cluster control or indicator by installing any combination of Booleans, strings, charts, graph scalars, arrays, or even other clusters into a *cluster shell*. You access the cluster shell from the **Controls** palette, as shown in the following illustration.



A new cluster shell has a resizable border and an optional label.

When you pop up in the empty element area, the **Controls** palette appears. You can place any element from the **Controls** palette in a cluster. You can drag existing objects into the cluster shell or deposit them directly inside by selecting them from the cluster pop-up menu. The cluster takes on the data direction (control or indicator) of the first element you place in the cluster, as do subsequently added objects. Selecting **Change To Control** or **Change To Indicator** from the pop-up menu of any cluster element changes the cluster and all its elements from indicators to controls or vice versa.

Operating and Configuring Cluster Elements

With the exception of setting default values and the **Change to Control** and **Change To Indicator** options, you configure controls and indicators the same way you configure controls and indicators that are not in a cluster.

Selecting Cluster Elements
Setting Cluster Default Values
Setting the Order of Cluster Elements
Moving or Resizing the Cluster

Selecting Cluster Elements

When in run mode, you can use the <Tab> key either to move the key focus between front panel controls, or between elements within a single cluster. Initially, the <Tab> key moves the key focus between front panel controls. To have it move between the elements within a specific cluster, first <Tab> to that cluster. Then use the <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key and the down arrow, which allows you to move between the cluster elements. If you want to return to tabbing between controls, use the <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key and the up arrow.

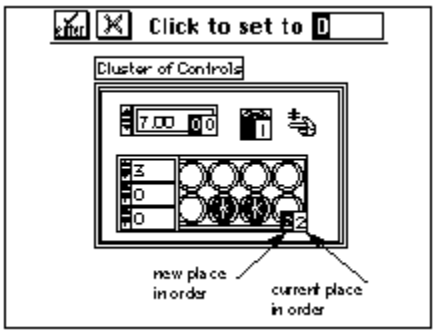
Setting Cluster Default Values

To set the default value of the entire cluster to the current value of each individual element, pop up on the frame and select the **Data Operations»Make Current Value Default** command from the cluster pop-up menu. To reset all elements to their individual default configurations, select the **Data Operations»Reinitialize To Default Values** command from the cluster pop-up menu.

To change the default value of a single element in a cluster, pop up on that particular element and select **Make Current Value Default**.

Setting the Order of Cluster Elements

Cluster elements have a logical order that is unrelated to their positions within the shell. The first object you insert in the cluster is element 0, the second is 1, and so on. If you delete an element, the order adjusts automatically. You can change the current order by selecting the **Cluster Order...** option from the cluster pop-up menu. The appearance of the element changes, as shown in the following illustration.




enter button


X button

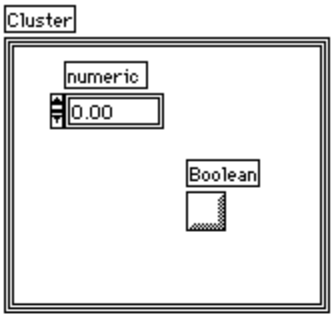
The white boxes on the elements show their current places in the cluster order. The black boxes show the elements new place in the order. Clicking on an element with the cluster order cursor sets the elements place in the cluster order to the number displayed inside the Tools palette. You change this order by typing a new number into that field. When the order is as you want it, click on the enter button to set it and exit the cluster order edit mode. Click on the X button to revert to the old order.

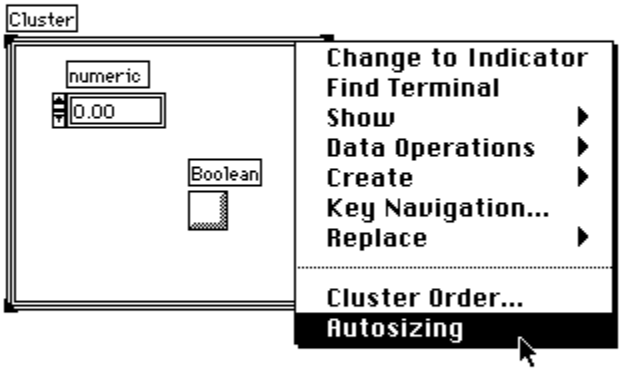
The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions in the block diagram. For more information on these functions, see the String Function Descriptions topic.

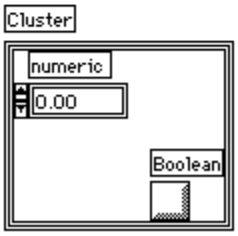
Moving or Resizing the Cluster

Cluster elements are not permanent components of clusters. That is, you can move or resize elements independently even when they are in the shell. To avoid inadvertently dragging them out, click on the shell when you want to move a cluster, and resize clusters from the shell border. If you inadvertently drag an element when you meant to drag the whole cluster, you can cancel the operation by dragging the element back into the shell or past the front panel window border before releasing the mouse button. You also cancel a resizing operation by dragging the border past the front panel window before releasing the mouse button.

You can shrink clusters to fit their contents by selecting Autosizing from the pop-up menu, shown in the following illustration.







Assembling Clusters

LabVIEW has three functions for assembling or building clusters. The Bundle and Bundle By Name functions assembles a cluster from individual elements or replaces individual elements with elements of the same type. The Array To Cluster function converts an array of elements into a cluster of elements.

Bundle Function
Bundle By Name Function
Array To Cluster Function

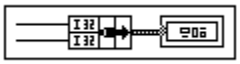
Bundle Function

Bundle function

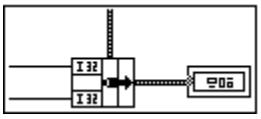
The Bundle function, which you obtain from the **Array & Cluster** palette of the **Functions** palette, appears on the block diagram with two element input terminals on the left side. You can resize the icon vertically to create as many terminals as you need. The element you wire to the top terminal becomes element 0 in the output cluster, the element you wire to the second terminal becomes element 1 in the output cluster, and so on.

As you wire to each input terminal, a symbol representing the data type of the wired element appears in the formerly empty terminal shown in the following illustration. You must wire all the inputs that you

create.

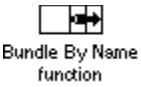


In addition to input terminals for elements on the left side, the Bundle function also has an input terminal for clusters in the middle as shown in the following illustration. You use this terminal to replace one or more elements of an existing cluster wire without affecting the other elements.



See Replacing Cluster Elements for an example of using this function to replace elements in a cluster.

Bundle By Name Function

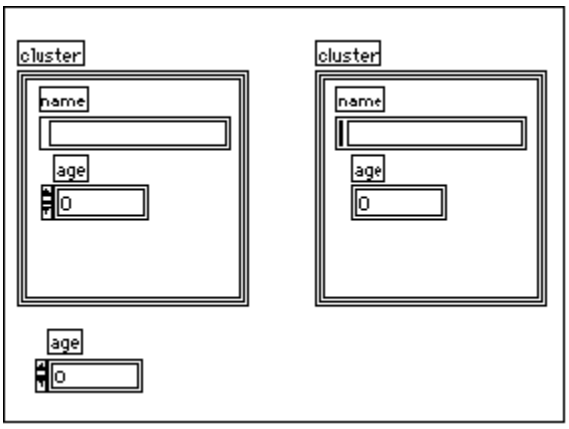


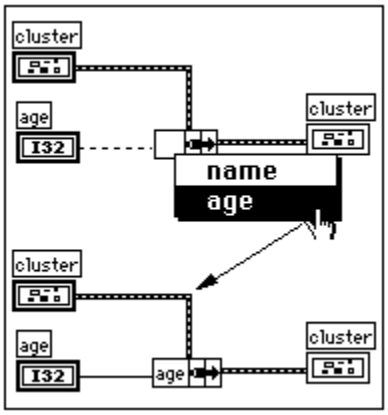
The Bundle By Name function, also in the **Array & Cluster** palette of the **Functions** palette, is similar to the Bundle function. Instead of referencing fields by position, however, you reference them by name. Unlike the Bundle function, you can access only the elements that you want to access. For each element you want to access, you need to add an input to the function.

Because the name does not denote a position within the cluster, you must wire a cluster to the middle terminal as well. You can only use the Bundle By Name function to replace elements by name, not to create a new cluster. You could, however, wire a data type cluster to the middle terminal, and then wire all of the new values as names to get this same behavior.

For example, suppose you have a cluster containing a string called Name and a number called Age. After placing the Bundle By Name function, you first need to connect an input cluster to the middle terminal of the Bundle By Name function.

After you have wired the middle terminal, you can select elements you want to modify by popping up on any of the left terminals of the Bundle By Name function. You then get a list of the names of the elements in the source cluster. After you select a name, you can wire a new value to that terminal to assign a new value to that element of the cluster. An example of this is shown in the following illustrations.

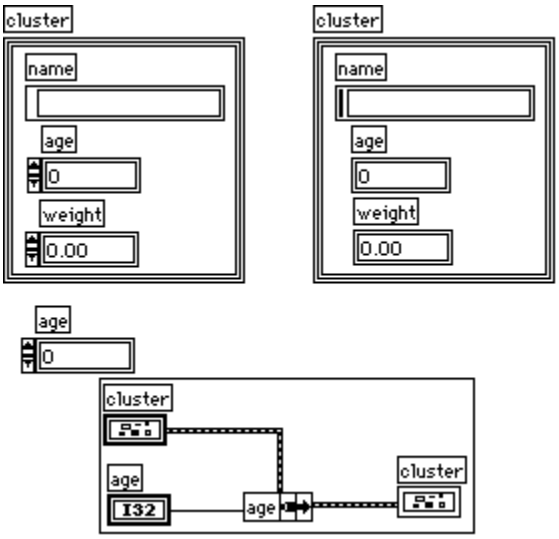




Notice that you can resize the Bundle By Name function to show as many or as few elements as you need. This is different from the Bundle function, which requires you to size it to have the same number of elements as are in the resulting cluster.

If you pop up on Bundle By Name when the cluster contains a cluster, you will find a pull to the side submenu that allows you to access individual elements of the subcluster by name, or select all elements.

The Bundle By Name function is useful when you are working with data structures that may change during the development process. If your modification involves the addition of a new element, or a reordering of the cluster, the change does not require changes to the Bundle By Name function, because the names are still valid. For example, if you modified the previous example by adding the new element and weight to the source and destination clusters, the VI is still correct. This example is shown in the following illustration.



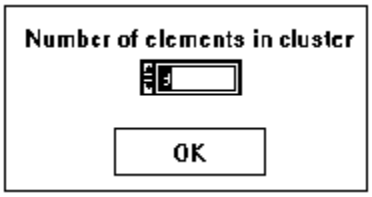
Bundle By Name is particularly useful in larger applications in conjunction with type definitions. By defining a cluster that may change as a type definition, any VI that uses that control can be changed to reflect a new data type by updating the type definitions file. If the VIs use only Bundle By Name and Unbundle By Name to access the elements of the cluster, you can avoid breaking the VIs that use the cluster.

See Type Definitions for more information on type definitions. Also see Unbundle By Name

Array To Cluster Function

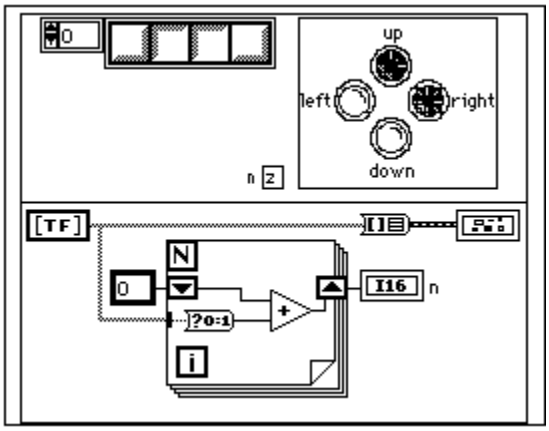
The Array To Cluster function, which you select from the **Conversion** palette of the **Functions** palette, converts the elements of a 1D array to elements of a cluster. This function is useful when you want to display elements within a front panel cluster indicator, but you want to manipulate the elements by index value on the block diagram.

Use the **Cluster Size...** option from the function pop-up menu to configure the number of elements in the cluster. The dialog box that appears is shown in the following illustration.



The function assigns the 0th array element to the 0th cluster element, and so on. If the array contains more elements than the cluster, the function ignores the remaining elements. If the array contains fewer elements than the cluster, the function assigns the extra cluster elements the default value for the element data type.

The illustration that follows shows how to display an array of data in a front panel cluster indicator. With this technique, you can organize the array elements on the front panel to suit your application. You could use a tabular array indicator instead, but that limits you to a fixed display formathorizontal with the lowest element at left and an attached index display.



Disassembling Cluster Elements

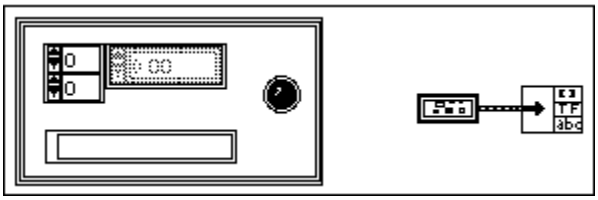
LabVIEW has three functions for disassembling clusters. The Unbundle and Unbundle By Name functions disassemble a cluster into individual elements, and the Cluster To Array function converts a cluster of identically typed elements into an array of elements of that type.

Unbundle Function
Unbundle By Name Function
Cluster To Array Function

Unbundle Function

The Unbundle function, which you obtain from the **Array & Cluster** palette of the **Functions** palette, has two element output terminals on the right side. You adjust the number of terminals with the Resizing tool the same way you adjust the Bundle function. Element 0 in the cluster order passes to the top output terminal, element 1 passes to the second terminal, and so on.

The cluster wire remains broken until you create the correct number of output terminals. Once the number of terminals is correct, the wire becomes solid. The terminal symbols display the element data types, as shown in the following illustration.

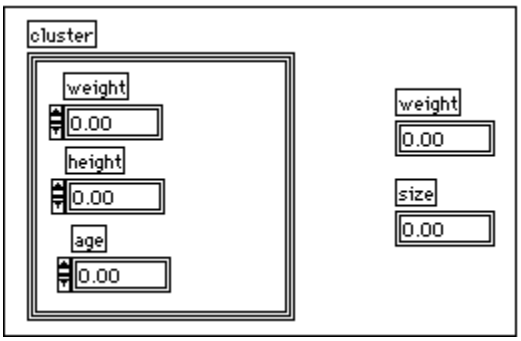


Unbundle By Name Function

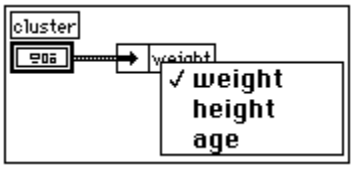
Unbundle By Name function

The Unbundle By Name function, also in the **Array & Cluster** palette of the **Functions** palette, is similar to the Unbundle function. Instead of referencing fields by position, however, you reference them by name. Unlike the Unbundle function, you can read only the elements that you want to read; you do not have to have one output for every cluster element.

For example, the following is a panel with a cluster of three elements, weight, height and age.



When connected to an Unbundle By Name function, you can pop up on an output terminal of the function and select an element that you want to read from the names of the components of the cluster, as shown in the following illustration.

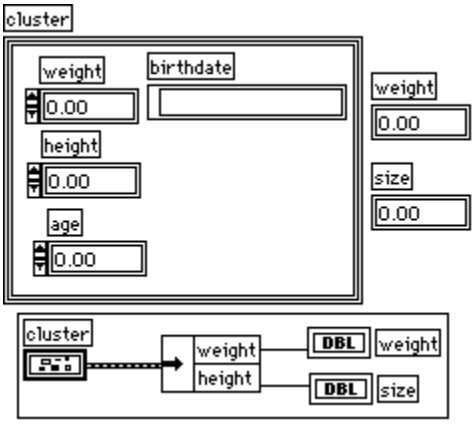


You can resize the function to read other elements, shown in the following illustration. Notice that you do not have to read all of the elements, and that you can read them in arbitrary order.

One of the advantages of the Unbundle By Name function over the Unbundle function is that it is not as

closely tied to the data structure of the cluster. The Bundle function must always have exactly the same number of terminals as it has elements. If you add or remove an element from a cluster connected to an Unbundle function, you end up with broken wires. With the Unbundle By Name function, you can add elements to and remove elements from the cluster without breaking the VI, as long as the elements were not referenced on the diagram.

For example, with the previous cluster, you could add a string for birth date, and the diagram would still be correct. This example is shown in the following illustration.



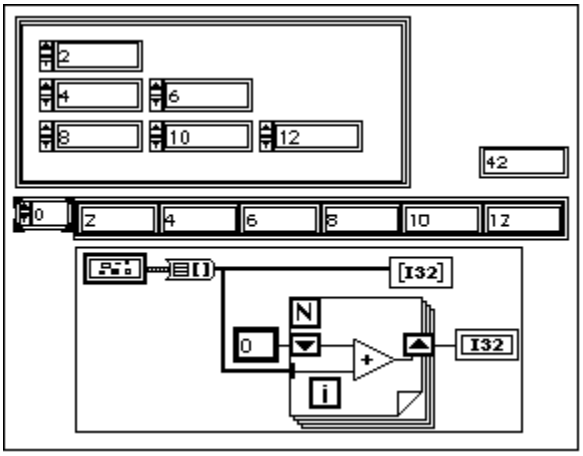
As with Bundle By Name, the Unbundle By Name function is particularly useful in larger applications in conjunction with type definitions. See Bundle By Name and Type Definitions, for more information.

Cluster To Array Function

The Cluster To Array function, which you obtain from the **Conversion** palette of the **Functions** palette, converts the elements of a cluster into a 1D array of those elements. The cluster elements must all be the same type. The array contains the same number of elements as the cluster.

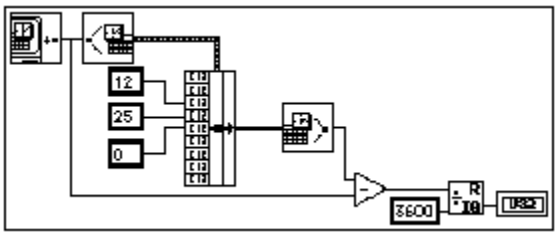
Cluster elements cannot be array elements. If you have a cluster of *N*-dimensional arrays that you want to convert to an *N*+1-dimensional array, you must unbundle the cluster elements with the Unbundle function and wire them to a Build Array function.

The Cluster To Array function is useful when you want to group a set of front panel controls in a particular order within a cluster but you want to process them as an array on the block diagram, as shown in the following illustration.



Replacing Cluster Elements

Sometimes you want to replace, or change the value of, one or two elements of a cluster without affecting the others. You can do this by unbundling a cluster and wiring the unchanging elements directly to a Bundle function along with the replacement values for the other element. The following example shows a more convenient method. This example computes the number of hours until New Year using the date-time cluster. Notice that because the cluster input terminal (middle terminal) of the Bundle function is wired, the only element input terminals that need to be wired are those with replacement values.



Graph and Chart Controls and Indicators

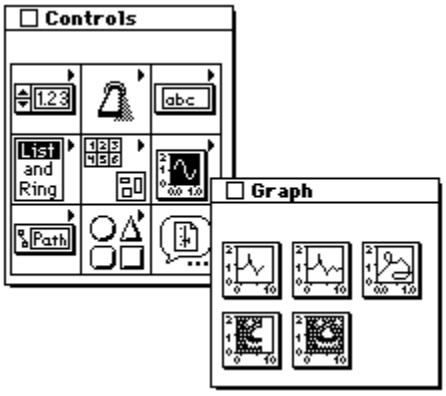
This topic describes how to create and use the graph and chart indicators in the **Graph** palette of the **Controls** palette.

Waveform and XY Graphs
Waveform Chart
Intensity Chart
Intensity Graph
Graph Cursors

A graph indicator is a two-dimensional display of one or more *plots*. The graph receives and plots data as a block. A chart also displays plots, but it receives the data and updates the display point by point or array by array, retaining a certain number of past points in a buffer for display purposes.

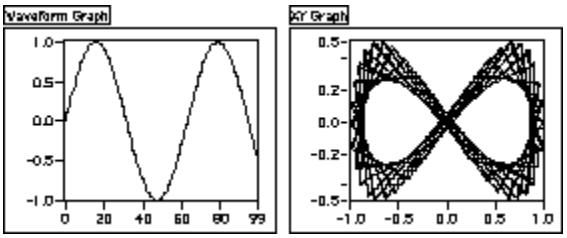
In addition to the information in this topic, you might find it helpful to study the graph and chart examples included with LabVIEW. These examples are located in `examples\general\graphs`.

LabVIEW has three kinds of graphs and two kinds of charts. These are shown in the following illustration starting at the top from left to right--waveform chart, waveform graph, XY graph, intensity chart, and intensity graph.



Waveform and XY Graphs

You can obtain a *waveform graph* and *XY graph* from the **Graph** palette of the **Controls** palette. Examples of these graphs are shown in the following illustration.



The waveform graph plots only single-valued functions with points that are evenly distributed with respect to the x-axis, such as acquired time-varying waveforms. The XY graph is a general-purpose, Cartesian graphing object you can use to plot multivalued functions such as circular shapes or waveforms with a varying timebase. Both graphs can display any number of plots.

Each of these graphs can accept several data types. This minimizes the extent to which you need to manipulate your data before displaying it. These data types are described in Creating a Single-Plot Graph and Creating a Multiplot Graph.

Graph Options describes some of the more advanced options of the graph.

Creating a Single-Plot Graph

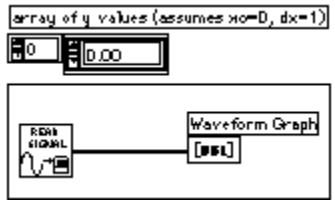
You can display single plots on a single waveform or XY graph.

Single-Plot Waveform Graph Data Types
Single-Plot XY Graph Data Types

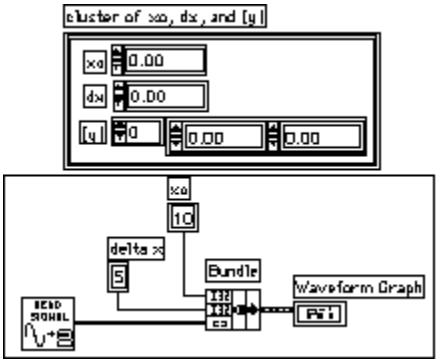
Single-Plot Waveform Graph Data Types

For single-plot graphs, the waveform graph accepts two data types. These data types are described in the following paragraphs.

The first data type that the waveform graph accepts is a single array of values. LabVIEW interprets the data as points on the graph and increments the points (starting at x=0) by 1. The following diagram illustrates how you might create this kind of data.



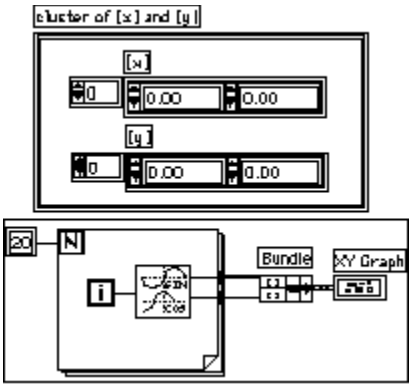
The second data type is a cluster of an initial x value, a Δx, and an array of y data. The following diagram illustrates how you might create this kind of data.



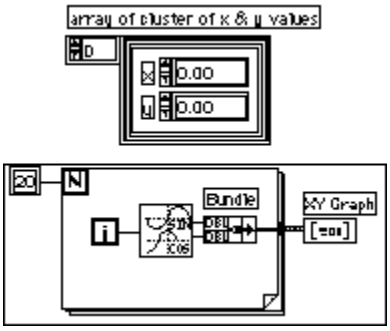
Single-Plot XY Graph Data Types

The XY graph accepts two data types for a single-plot graph. These data types are described in the following paragraphs.

The first data type that the XY graph accepts is a cluster containing an x array and a y array. The following diagram illustrates how you might create this kind of data.



The second data type is an array of *points*, where a point is a cluster of an x value and a y value. The following diagram illustrates how you might create this kind of data.



Creating a Multiplot Graph

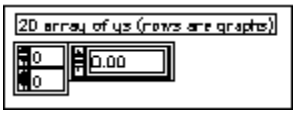
You can display multiple plots on a single waveform or XY graph. For the most part, you use arrays of the data types described in Creating a Single-Plot Graph to describe multiple plots for display in a single graph. Because LabVIEW does not allow arrays of arrays, in a case where creating an array of a single plot data type would produce an array of arrays, you can use either 2D arrays or arrays of clusters of arrays.

Multiplot Waveform Graph Data Types
Multiplot XY Graph Data Types

Multiplot Waveform Graph Data Types

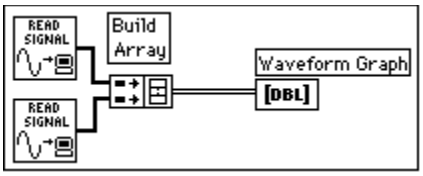
The multiplot waveform graph accepts five data types, which are described in the following paragraphs.

The first data type that a multiplot graph waveform accepts is a two-dimensional array of values, where each row of the data is a single plot. LabVIEW interprets this data as points on the graph, with the points starting at x=0 and incremented by 1.

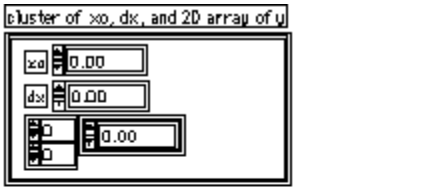


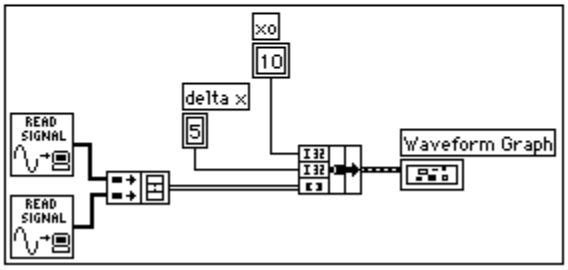
If you select the **Transpose Array** option from the graph pop-up menu, each column of data is treated as a plot. This is particularly useful when sampling multiple channels from a data acquisition board, because that data is returned as 2D arrays, with each channel stored as a separate column.

The following example shows how a graph could be used to display two signals, where each signal is a separate row of a two-dimensional array.

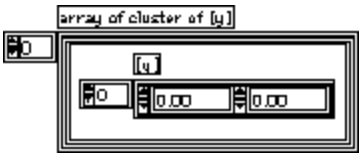


The second data type is a cluster of an x value, a Δx value, and a two-dimensional array of y data. The y data is interpreted as described for the previous data type. This data type is useful for displaying multiple signals that were all sampled at the same regular rate. The following diagram illustrates how you might create this kind of data.



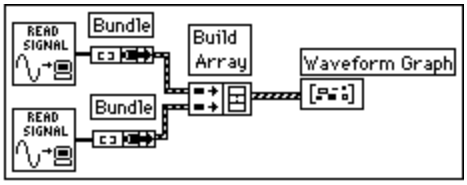


The third data type is an array of clusters of an array of y data. LabVIEW interprets this data as points on the graph, with the points starting at x=0 and incremented by 1.



Use this data structure instead of a two-dimensional array if the number of elements in each plot is different. For example, you might need to sample data from several channels, but not for the same amount of time from each channel. You would then use this data structure because each row of a two-dimensional array must have the same number of elements, but the number of elements in the interior arrays of an array of clusters can vary.

The following illustration shows how to create the appropriate data structure from two arrays.



To see an example of this concept, look at the Waveform Graph VI in the `examples\general\graphs\gengraph.llb` directory.

Another way to make arrays elements of an array of clusters is to use the build cluster array function.

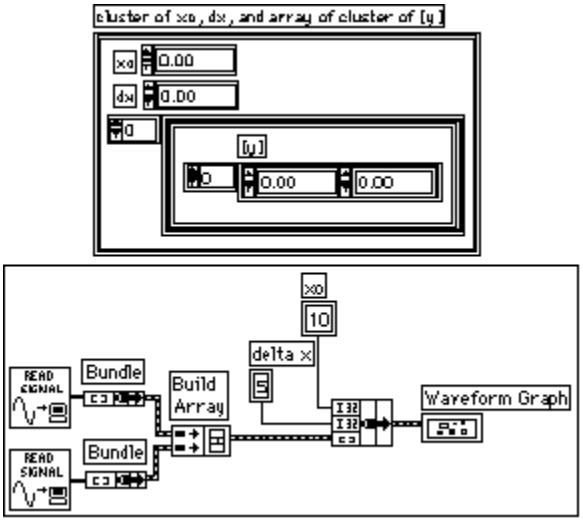
The fourth data type is a cluster of an initial x value, a Δx value, and an array of clusters of an array of y data.

Use this data structure instead of a two-dimensional array if the number of elements in each plot is different. For example, you might need to sample data from several channels, but not for the same

amount of time from each channel. You would then use this data structure because each row of a two-dimensional array must have the same number of elements, but the number of elements in the interior arrays of an array of clusters can vary. The following diagram illustrates how you might create this kind of data.

Notice that the arrays are bundled into clusters using the Bundle function, and the resulting clusters built into an array with the Build Array function. You could use the Build Cluster Array, which creates arrays of clusters of specified inputs, instead.

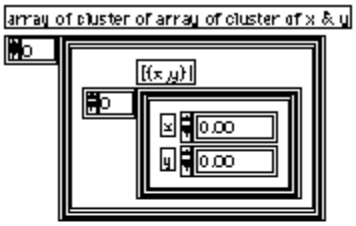
The fifth data type is an array of clusters of an x value, a Δx value, and an array of y data. This is the most general of the waveform graph multiplot data types, because it allows you to specify a unique starting point and increment for the x-axis of each plot. The following diagram illustrates how you might create this kind of data.

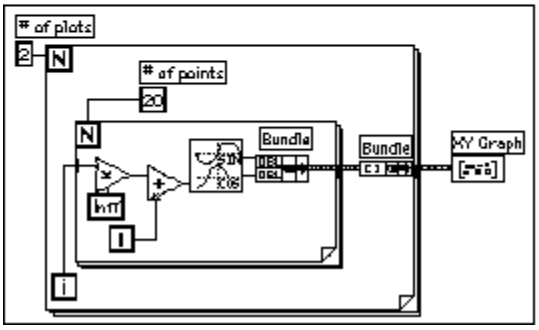


Multiplot XY Graph Data Types

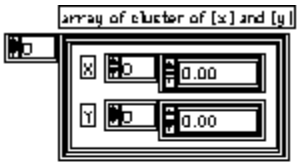
The XY graph accepts two multiplot data types, as described in the following paragraphs. Both of these data types are arrays of clusters of the single plot data types described previously.

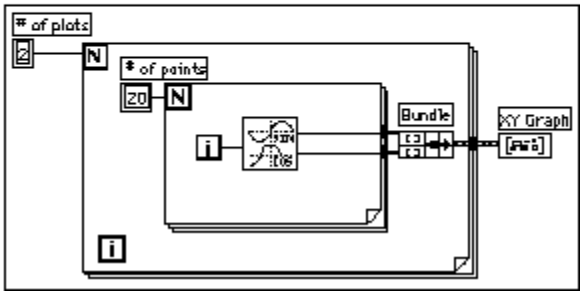
The first data type is an array of clusters of plots, where a plot is an array of points. A point is defined as a cluster containing an x and y value. The following diagram illustrates how you might create this kind of data.





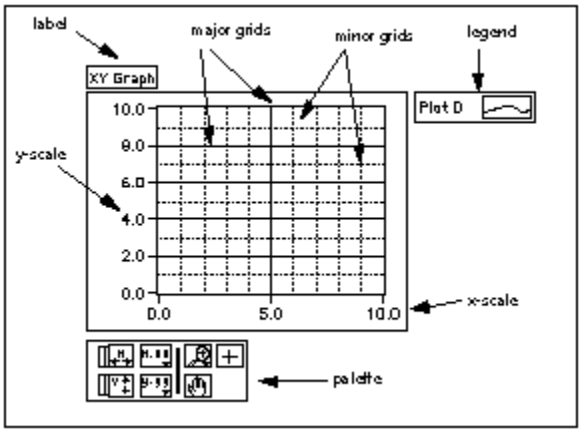
The second data type is an array of plots, where a plot is a cluster of an x array and a y array. The following diagram illustrates how you might create this kind of data.





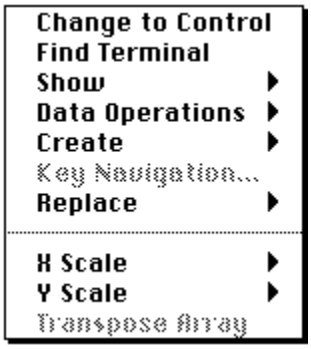
Graph Options

Both graphs have optional parts that can be shown or hidden using the **Show** submenu of the graph pop-up menu. These options include a legend, from which you can define the color and style for a given plot; a palette, that you use to change scaling and format options while the VI is running; and a Cursor palette that you use to display multiple cursors. Following is a picture of a graph showing all of these optional components except for the Cursor palette, which is illustrated in Graph Cursors.



Marker Spacing
Formatting
Autoscale
Loose Fit
Panning and Zooming Options
Using the Legend
Graph Cursors

Graphs have many options you can use to customize your data display. The graph pop-up menu is shown in the following illustration. **Transpose Array** is available with the waveform graph only.

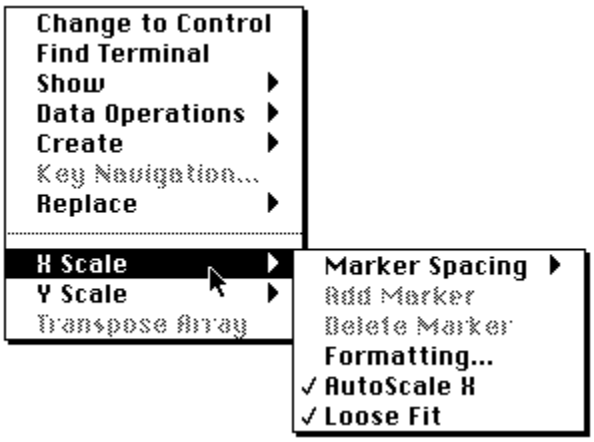


Graphs automatically adjust their horizontal and vertical scales to reflect the array of points you wire to them. You can turn this autoscaling feature on or off using the **Autoscale X** and **Autoscale Y** options from the **Data Operations** or the **X Scale/Y Scale** submenus of the graphs pop-up menu. You can also control these autoscaling features from the graphs palette, as described in Autoscale. Autoscaling on is the default setting for LabVIEW. However, Autoscaling may cause the graph to run more slowly.

You can change the horizontal or vertical scale directly using the Operating or Labeling tool, just as you can with any other LabVIEW control or indicator. LabVIEW sets point density automatically.

The **Data Operations** submenu of the graph pop-up menu includes a **Smooth Updates** option that uses an offscreen buffer to minimize flashing. This feature may cause the graph to run more slowly, depending on the computer and video system you use.

The X and Y scales each have a submenu of options, as shown in the following illustration.

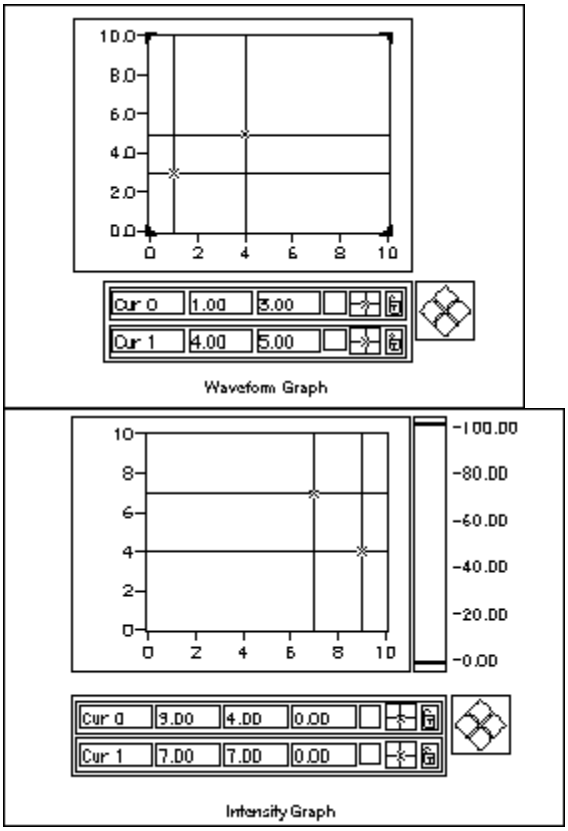


Graph Cursors

For each graph, you can display a Cursor palette that you use to put cursors on the graph. You can label the cursor on the plot. You can also use a cursor as a marker. When you use a cursor as a marker, you lock the cursor to a data plot so that the cursor follows the data.

Cursors can be set and read programmatically using the Attribute Node. 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 that a graph can have.

Following are illustrations of a waveform graph and an intensity graph with the Cursor palette displayed.



Each cursor for a graph has the following parts.

A label

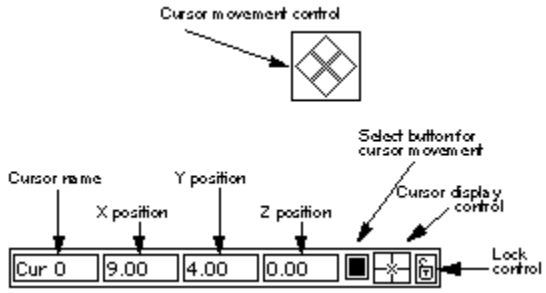
X and Y coordinates, and Z coordinate, if applicable

A button that marks the plot for movement with the plot cursor pad

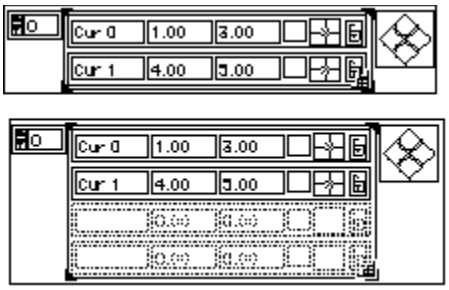
A button that controls the look of the cursor

A button that determines whether the cursor is locked to a plot or can be moved freely

These parts are shown in the following illustration.

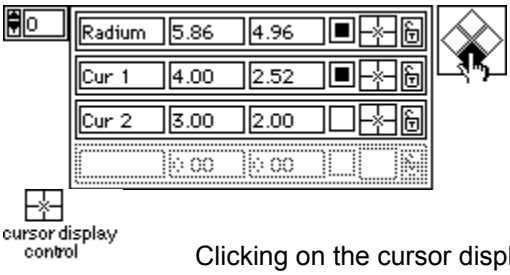


The Cursor palette behaves like an array. You can stretch it to display multiple cursors, and you can use the index control to view other cursors in the palette. Use the **Show** options on the pop-up menu to display the index control.

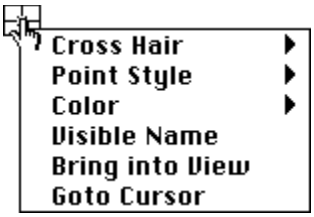


To delete a cursor, you must select it using the **Start Selection** and **End Selection** options on the **Data Operations** pop-up menu, and then cut the cursor with the **Cut** option on the same menu. See Selecting Array Cells for more information.

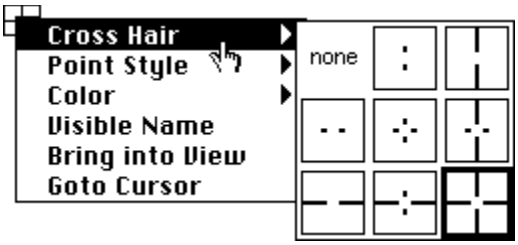
You can move a cursor on a graph or chart by dragging it with the Operating tool, or by using the cursor movement control. Clicking the arrows on the cursor movement control causes all cursors selected to move in the specified direction. You select cursors either by moving them on the graph with the Operating tool, or by clicking on the select button for a given cursor. In the following example, the top two cursors would be moved vertically downwards.



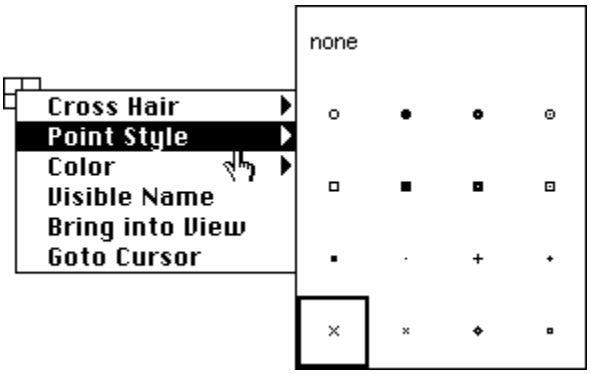
Clicking on the cursor display control with the Operating tool displays a pop-up menu that you can use to control the look of the cursor and the visibility of the cursor name on the plot. This pop-up menu is shown in the following illustration.



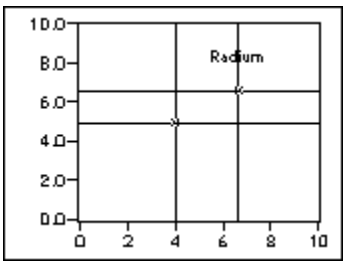
From this menu you can select the style of the cross hairs. The cross hairs can consist of a vertical and/or horizontal line extending to infinity, a smaller vertical and/or horizontal line centered about the cursor location, or no cross hairs, as shown in the following illustration.



You can also choose the style of point to use for marking the cursor location and the color for the cursor as shown in the following illustration.



Select the **Visible Name** option from this menu to make the cursor name visible on the plot, as shown in the following illustration.



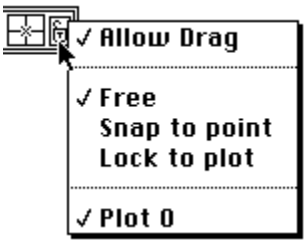
Selecting **Bring into View** moves the cursor back into the displayed region of the graph. This is helpful when the cursor has moved out of visible range. Selecting this option changes the (*x,y*) coordinate position of the cursor.

Selecting **Goto Cursor** moves the displayed region of the graph so that the cursor is visible. The cursor position remains constant, but the scales change to include the cursor selected. The size of the displayed region also stays constant. This feature is helpful when the cursor is used to identify a point of interest in the graph, such as a minimum or a maximum, and you want to see that point.

lock button

You can use the last button for each cursor to lock a cursor onto a given plot. Clicking on

the lock button gives you a pop-up menu that you can use to lock the cursor to a specific plot. If you lock the cursor onto a plot, the button changes to a closed lock. This pop-up menu is shown in the following illustration.



The **Allow Drag** option determines whether you can move the cursor with the mouse. If **Allow Drag** is selected, you can move, or drag the cursor. If Allow Drag is deselected, you cannot move the cursor around on the plot. If you have **Allow Drag** selected, the options below the dotted line of the menu determine how you can move the cursor with the mouse.

Select **Free** if you want to place or move the cursor anywhere on the graph. Select **Snap to Point** if you want the cursor to always attach itself to the nearest point on any plot. Select **Lock to Plot** to attach the cursor to a specific point. The first time you select **Lock to Plot**, the cursor attaches itself to the first point on the plot. After freeing the locked cursor and moving it to any new position, selecting **Lock to Plot** moves the cursor to the last location of the locked cursor.

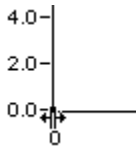
Below the second dotted line of this menu is a list of the plots you can lock to (for example, Plot 0, Plot 1, Plot 2, and so on).

A large number of options are available for creating, controlling, and reading cursors or markers programmatically using the Attribute Node for a graph. See Attribute Nodes for further information.

Marker Spacing

By default, marker values for x and y scales are uniformly distributed. If you prefer, you can specify marker locations anywhere on the x or y scale. This is useful for marking a few specific points on a graph (such as a set point or threshold).

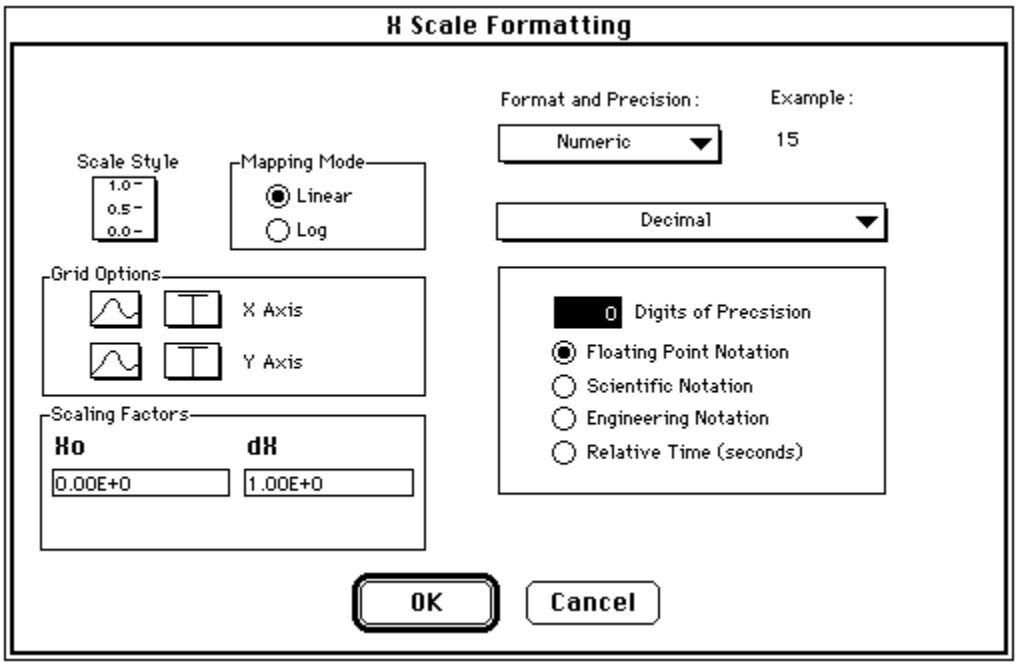
If you want non-uniform marker distribution, choose **X or Y Scale» Marker Spacing»Arbitrary Markers** from the scales pop-up menu. After making this selection, if you move the Operating tool over a tick mark, the cursor changes to the double arrow cursor shown in the following illustration. You can now create a new marker by dragging the existing tick mark with the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key selected, or you can move the existing tick mark anywhere on the scale you want by dragging it.



You can add or delete a marker by popping up on the graph or the scale and selecting **Add Marker** or **Delete Marker**. Once a marker is created, you can type a number in the marker to change its location.

Formatting

Click on **Formatting…** to bring up the dialog box shown in the following illustration.



Select the options in this dialog box to set the following graph properties.

Scale Style--Use this option to select whether you want major and minor tick marks for the scale. Major tick marks are points corresponding to scale labels and minor tick marks are interior points between labels. You also use this palette to select whether you want the markers for a given axis to be visible.

Mapping Mode--Use this option of the scale menus to select whether the scale should display data using a linear or a logarithmic scale.

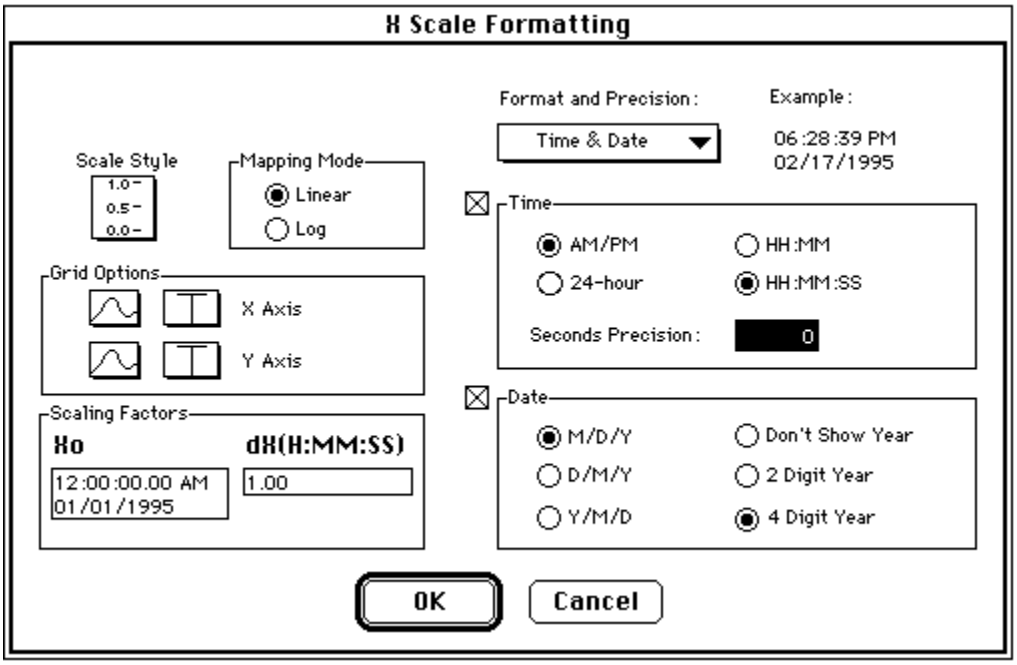
Grid Options--Clicking on these options brings up a palette of grid types to select whether you want no gridlines, gridlines only at major tick mark locations (points corresponding to scale labels) or major and minor tick marks (minor tick marks are interior points between labels). The control next to the grid popup is a color ring you can use to select the color for the gridlines.

Scaling Factors--You use the scaling factors to specify the initial value and spacing between points on a waveform chart or graph, or along the scales of an intensity chart or graph. You can also use these scaling factors to scale your data for display. For example, if your data was binary sampled data in a range of -2,048 to 2,047, you could scale this data to its appropriate voltage values of -10 to 10 by specifying an additive offset of 2,038, and a scaling factor of 4,096/20 = 204.8.

Format & Precision--At the top of the dialog box is a pop-up menu that lets you select whether the X or Y scale is formatted for numerics or for time and date.

If you select **Numeric** formatting, you can choose whether the notation is floating-point, scientific, engineering, or relative time in seconds; whether the notation is (decimal, unsigned decimal, hexadecimal, octal, and binary); and you can select the precision, that is how many digits are displayed to the right of the decimal point, from 0 through 20. The precision you select affects only the display of the value; the internal accuracy still depends on the representation. Examples are shown in the dialog box as you select each combination of options.

If you select **Time & Date** formatting, the dialog box changes, as shown in the following illustration.



You can format for either absolute time or date, or both. If you enter only time or only date, LabVIEW infers the unspecified components. If you do not enter time, LabVIEW assumes 12:00 a.m. If you do not enter date, it assumes the previous date value. If you enter date, but the scale is not in a date format, LabVIEW assumes the month, day, and year ordering based on the settings in the Preferences dialog box. If you enter only two digits for the year, LabVIEW assumes the following: any number less than thirty-eight is in the twenty-first century, otherwise the number is in the twentieth century.

Though absolute time is displayed as a time and date string, it is represented internally as the number of seconds since 12:00 AM January 1, 1904, Greenwich Mean Time. LabVIEW keeps track of these components internally.

Note: When a scale is in absolute time format, you always have the option to enter time, date, or time and date. If you do not want LabVIEW to assume a date, use relative time.

Note the examples at the top right of the dialog box, which change as you make selections.

The valid range for time and date differs across computer platforms as follows:

(Windows) 12:00 a.m. Jan. 2, 1970 - 12:00 a.m. Feb. 4, 2106

(Windows NT) 12:00 a.m. Jan. 2, 1970 - 12:00 a.m. Jan. 17, 2038

(Macintosh) 12:00 a.m. Jan. 2, 1904 - 12:00 a.m. Jan. 2, 2040

(UNIX) 12:00 a.m. Dec. 15, 1901 - 12:00 a.m. Jan. 17, 2038

These ranges may be up to a few days wider depending on your time zone and whether daylight saving time is in effect.

Autoscale

Use **AutoScale (X** or **Y)** to turn the autoscaling option on or off.

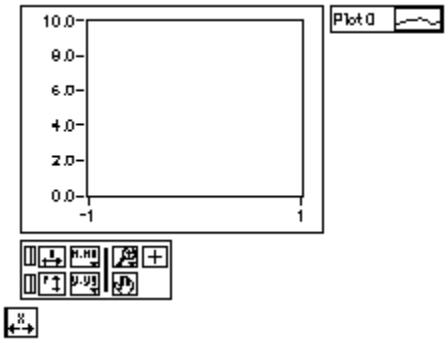
Loose Fit

Normally LabVIEW rounds the end markers of the scale to whole numbers. If you want the scales to be set to exactly the range of the data, turn off the **Loose Fit** option in the graph popup menu. With a loose

fit, the numbers are rounded to a multiple of the increment used for the scale. For example, if the markers increment by 5, then the minimum and maximum values are set to a multiple of 5.

Panning and Zooming Options

The **Graph** palette is included with any graph you drop onto the front panel. This palette has controls for panning (scrolling the display area of a graph) and for zooming in and out of sections of the graph. A graph with its accompanying Graph palette is shown in the following illustration.



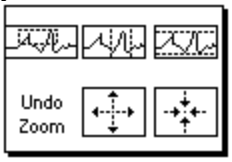

If you press the x autoscale button, shown at the left, LabVIEW autoscales the X data of the graph. If you press the y autoscale button, shown at the left, LabVIEW autoscales the Y data of the graph. If you want the graph to autoscale either of the scales continuously, click on the lock switch, shown at the left, to lock autoscaling on.


The scale format buttons, shown left, give you run-time control over the format of the X and Y scale markers respectively.
You use the remaining three buttons to control the operation mode for the graph.


Normally, you are in standard operate mode, indicated by the plus or crosshatch. In operate mode, you can click in the graph to move cursors around.


If you press the Panning tool, shown to the left, you switch to a mode in which you can scroll the visible data by clicking and dragging sections of the graph.


If you press the Zoom tool, shown at the left, you can zoom in on a section of the graph by dragging a selection rectangle around that section. If you click on the Zoom tool, you get a pop-up menu you can use to choose some other methods of zooming. This menu is shown in the following illustration.



A description of each of these options follows.


Zoom by rectangle.


Zoom by rectangle, with zooming restricted to x data (the y scale remains unchanged.


Zoom by rectangle, with zooming restricted to y data (the z scale remains unchanged).

Undo last zoom. Resets the graph to its previous setting.


Zoom in about a point. If you hold down the mouse on a specific point, the graph continuously zooms in until you release the mouse button.

 Zoom out about a point. If you hold down the mouse on a specific point, the graph continuously zooms out until you release the mouse button.

Note: For the last two modes, zoom in and zoom out about a point, `<Shift>`-clicking zooms in the other direction.

Using the Legend

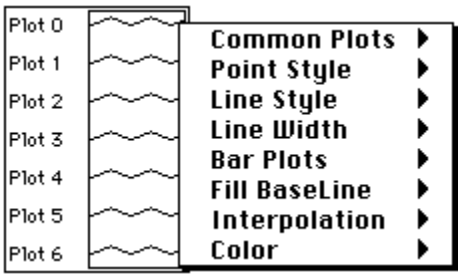
The graph uses a default style for each new plot unless you have created a custom plot style for it. If you want a multiplot graph to use certain characteristics for specific plots (for instance, to make the third plot blue), you can set these characteristics using the legend, which can be shown or hidden using the **Show** submenu of the graph pop-up menu. You can also specify a name for each plot using the legend.

When you select **Legend**, only one plot appears. You can create more plots by dragging down a corner of the legend with the Resizing tool. After you set plot characteristics, LabVIEW assigns those characteristics to the plot, regardless of the whether the legend is visible. If the graph receives more plots than are defined in the legend, LabVIEW draws them in default style.

When you move the graph body, the legend moves with it. You can change the position of the legend relative to the graph by dragging the legend to a new location. Resize the legend on the left to give labels more room or on the right to give plot samples more room.

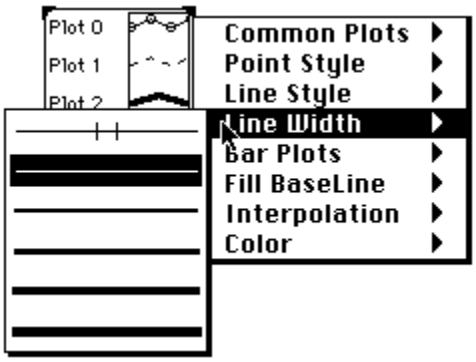
By default, each plot is labeled with a number, beginning with zero. You can modify this label the same way you modify other LabVIEW labels. To the right of the plot label is the *plot sample*. Each plot sample has its own pop-up menu to change the plot, line, color, and point styles of the plot. The array of points you wire to the graph are displayed with the characteristics you assign it in the graph legend.

The plot sample pop-up menu is shown in the following illustration.



The **Common Plots** option lets you easily configure a plot for any of six popular plot styles, including a scatter plot, a bar plot, and a fill to zero plot. Options in this subpalette are preconfigured for the point, line, and file styles in one step.

The **Point Style**, **Line Style**, and **Line Width** options display styles you can use to distinguish a plot. The line width subpalette offers widths thicker than the default 1 pixel, as well as the hairline option. The latter option has no effect on the screen display, but will print a very thin line if the printer and print mode support hairline printing.

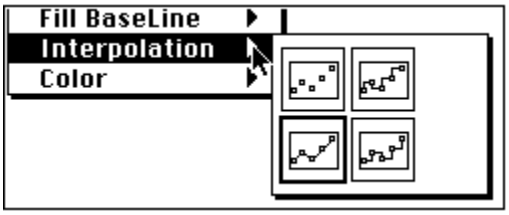


Note: In Windows 3.*x*, Windows 95, and Windows NT, wide pens can only be in the solid style.

The **Bar Plots** option allows you to choose among vertical bars, horizontal bars, or no bars at all.

The **Fill Baseline** option sets what the baseline fills to. Zero fills from your plot to a baseline generated at 0. Infinity fills from your plot to the positive edge of the graph. -Infinity fills from your plot to the negative edge of the graph. The bottom portion of this menu lets you select a specific plot of this graph to fill to.

The **Interpolation** option brings up the palette shown in the following illustration, in which you choose how LabVIEW draws lines between plotted points. The first option does not draw a line, making it suitable for a scatter plot. The option at the bottom left draws a straight line between plotted points. The two stepped options, which link points with a right-angled elbow, are useful for creating histogram-like plots. The option at the top right plots the y-axis first, while the option at the bottom right plots the x-axis first.



The **Color** option displays the palette for selecting the plot color. You can also color the plots on the legend with the Color tool, and you can change the plot colors while running the VI.

To change the properties of highlighted text, use the Font ring in the toolbar.

Waveform Chart

The waveform chart is a special type of numeric indicator that displays one or more plots. Charts are different from graphs in that charts retain previous data, up to a maximum which you can define. New data is appended to the old data, letting you see the current value in context with previous data.

For an example of a waveform chart, see `examples\general\graphs\charts.llb`.

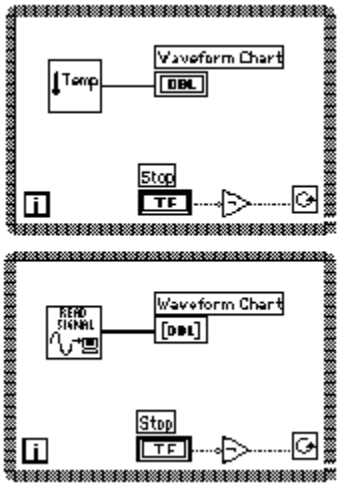
Waveform Chart Data Types
Waveform Chart Options

Waveform Chart Data Types

You can pass charts either a single value or multiple values at a time. As with the waveform graph, each value is treated as part of a uniformly spaced waveform, with each point spaced one point from the previous one along the x-axis.

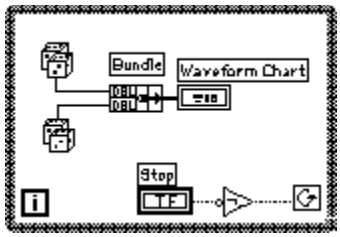
You can pass either a single scalar value or an array of multiple values to the chart. The chart treats these inputs as new data for a single plot.

Following are diagrams illustrating how you can use the chart for each of these kinds of data.

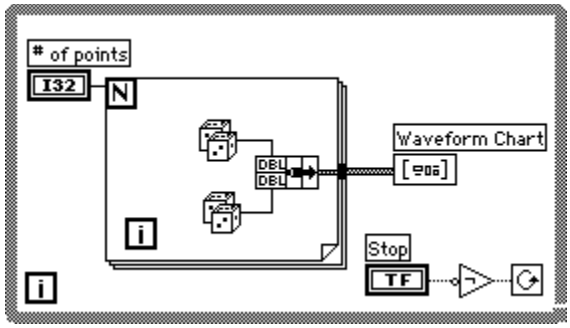


You get the best performance from a chart when you pass it multiple points at a time, because the chart has to be redrawn only once for each waveform instead of once for each point.

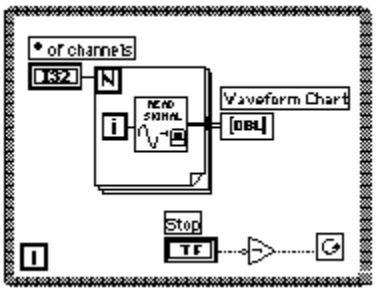
You can pass data for multiple plots to a waveform chart in several ways. The first method is to bundle the data together into a cluster of scalar numerics, where each numeric represents a single point for each of the plots. An example of this is shown in the following illustration.



If you want to pass multiple points for plots in a single update, you can wire an array of clusters of numerics to the chart. Each numeric represents a single point for each of the plots. An example of this is shown in the following illustration.

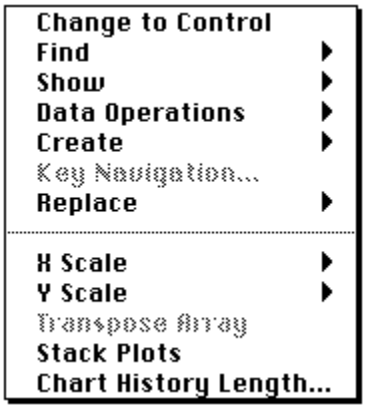


If the number of plots you want to display cannot be determined until runtime, or you want to pass multiple points for plots in a single update, then you can wire a two-dimensional array of data to the chart. As with the waveform graph, rows are normally treated as new data for each plot. You can use the **Transpose Array** option of the waveform chart pop-up menu to treat columns as new data for each plot.



Waveform Chart Options

The chart has most of the same features as the graph, including the legend and palette, and they work the same way. (See Graph Options for more information.) The waveform chart does not have support for cursors. The following illustration shows the chart pop-up menu.



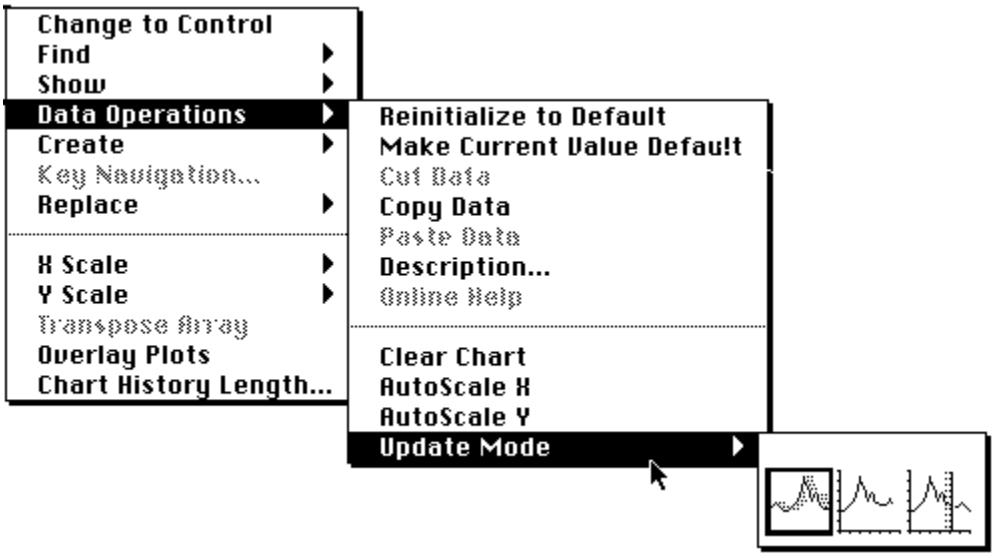
With the **Show** submenu of the chart pop-up menu, you can show or hide optional digital display(s) and a scrollbar. The **Digital Display** option displays the latest value being plotted. The last value passed to the chart from the diagram is the latest new value for the chart. There is one digital display per plot.

You can view past values contained in the buffer by scrolling the x-axis to a range of previously plotted values using the scrollbar, or by changing the x scale range to view old data.

The chart has a limit to the amount of data it can remember to avoid running out of memory. When the chart reaches that limit, the oldest point(s) are thrown away to make room for new data. The default size of this buffer is 1,024 points. You can change the length of this buffer using the **Chart History Length...** option from the chart pop-up menu.

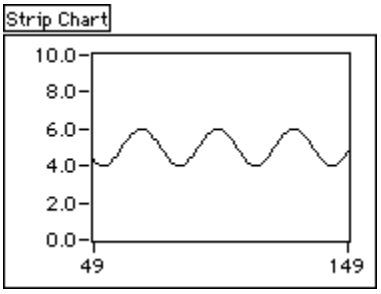
Chart Update Modes

To change the way the chart behaves when new data is added to the display, use the **Update Mode** option from the **Data Operations** submenu of the charts pop-up menu, shown in the following illustration.

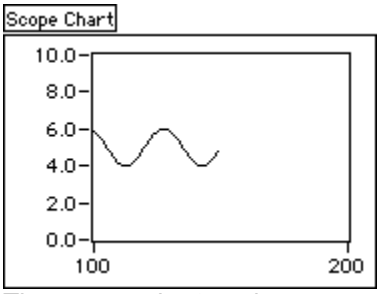


The three options--strip chart, scope chart, and sweep chart--are illustrated in the following pictures, and described in the subsequent paragraphs. The default mode is strip chart.

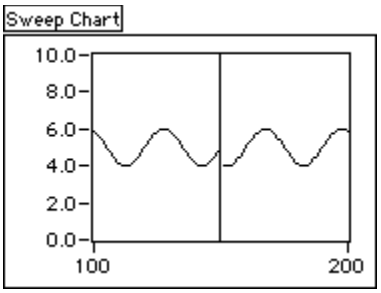
The *strip chart* mode has a scrolling display similar to a paper tape strip chart recorder. As each new value is received, it is plotted at the right margin, and old values shift to the left.



The *scope chart* mode has a retracing display similar to an oscilloscope. As each new value is received, it is plotted to the right of the last value. When the plot reaches the right border of the plotting area, the plot is erased and plotting begins 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.

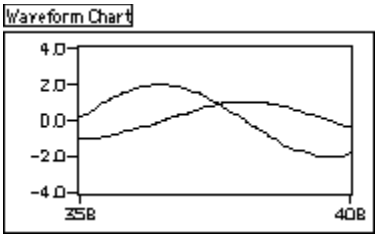


The *sweep chart* mode acts much like the scope chart, but it does not blank when the data hits the right border. Instead, a moving vertical line marks the beginning of new data and moves across the display as new data is added.

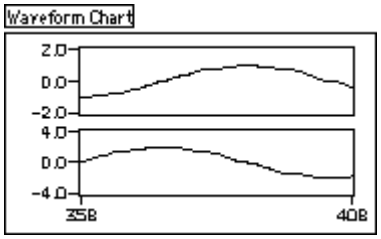


Stacked Versus Overlaid Plots

By default, the chart displays multiple plots by overlaying one on top of the other, like graphs drawn on the same grid. An example of overlaid plots is shown in the following illustration.



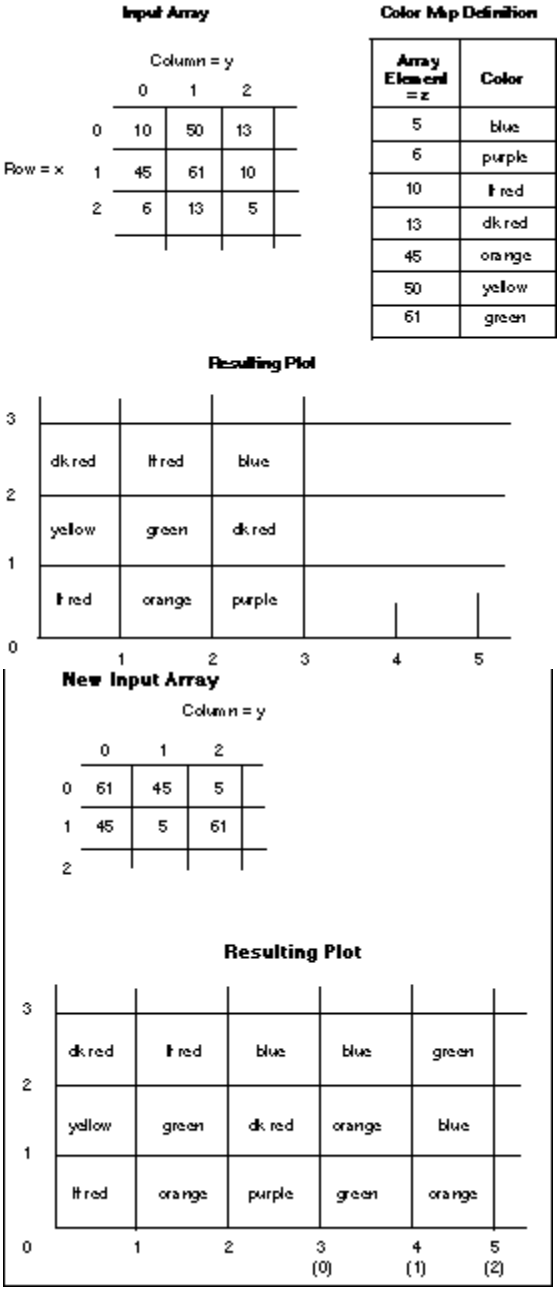
You can alternatively display multiple plots by stacking one above the other, with a different Y scale for each plot, by selecting the **Stack Plots** option from the chart pop-up menu. If you do this, then each charts Y scale can have a different range. An example of stacked plots is shown in the following illustration.



When you input data to the chart as a cluster, LabVIEW automatically stacks the correct number of plot displays. When you input data as a 2-D array, you must create the correct number of plot displays using the Legend, available in the **Show** submenu of the pop-up menu. As you enlarge the Legend display to increase the number of plots, the number of stacked plot displays increases to match.

Intensity Chart

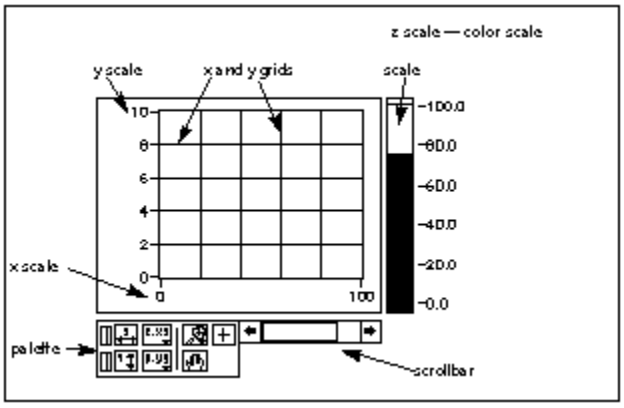
The intensity chart is a way of displaying three dimensions of data on a two-dimensional plot by placing blocks of color on a Cartesian plane. The intensity chart accepts a two-dimensional array of numbers. Each number in the array represents a specific color. The indices of an element in the two-dimensional array set the plot location for this color. The following illustration shows the concept of the intensity chart operation.



You can define the colors for the intensity chart interactively, by using the color scale, or you can define them programmatically through the chart Attribute Node. Defining the Color Mapping explains the procedure for assigning a color to a number.

The array indices correspond to the lower left vertex of the block of color. The block has a unit area, as defined by the array indices. The intensity chart can display up to 256 discrete colors. See Intensity Chart Options for more information.

After a block of data has been plotted, the origin of the Cartesian plane is shifted to the right of the last data block. When new data is sent to the intensity chart, the new data appears to the right of the old data, as shown in the following illustration.



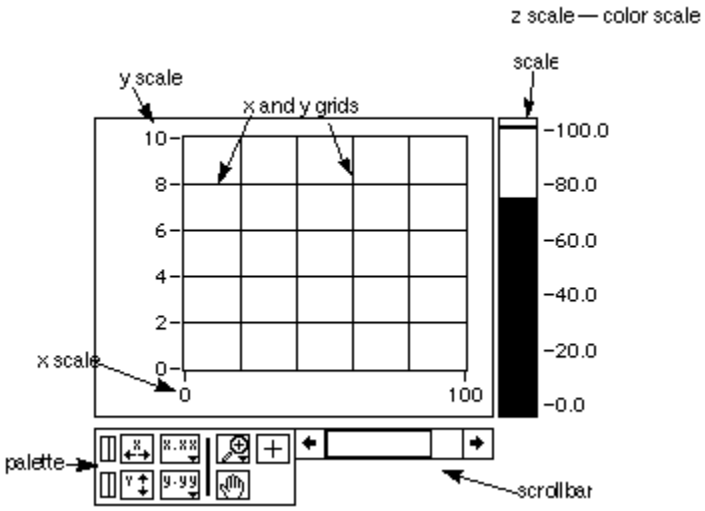
When the chart display is full, the oldest data scrolls off the left side of the chart.

See examples of intensity charts in `examples\general\graphs\intgraph.llb`.

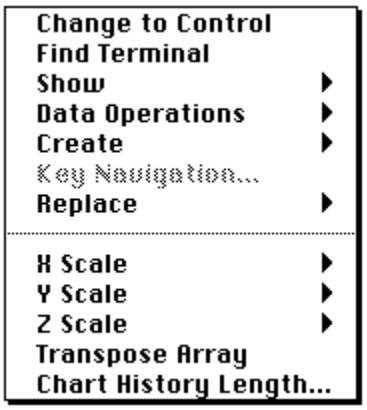
Intensity Chart Options

The intensity chart shares many of the optional parts of the other charts, most of which can be shown or hidden from the **Show** submenu of the graph pop-up menu. These options include a palette you use to change scaling and format options while the VI is running. In addition, because the intensity chart has a third dimension (color), a scale similar to a color ramp control is used to define the range and mappings of values to colors.

Following is a picture of a graph showing all of these optional components except for the Cursor palette, which is discussed in Graph Cursors.



Intensity charts have many options you can use to customize your data display. The intensity chart pop-up menu is shown below in the following illustration.



Most of these options are identical to the options for the waveform chart. With the **Show** menu, you can show and hide the z scales color scale. The **X Scale** and **Y Scale** menus are identical to the corresponding menus for the waveform chart.

The intensity chart maintains a history of data from previous updates. You can configure this buffer by selecting **Chart History Length...** from the chart pop-up menu. The default size for an intensity chart is 128 points. Notice that the intensity chart display can be very memory intensive. For example, to display a single precision chart with a history of 512 points and 128 y values requires 512 * 128 * 4 bytes (size of a single), or 256 kilobytes. If you want to display large quantities of data on an intensity chart, make sure enough memory is available to LabVIEW.

The intensity chart supports the standard chart update modes of strip chart, sweep chart, and scope chart. As with the waveform chart, you select the update mode from the **Data Operations** menu.

Defining the Color Mapping

You can set the color mapping interactively in the same way that you define the colors for a color ramp numeric control. See the description of the color ramp for more details.

You can set the colors used for display programmatically using the Attribute Node in two ways. First, you can specify the value-to-color mappings to the Attribute Node in the same way you do it with the color scale. For this method, you specify the **Z Scale Info: Color Array** attribute. This attribute consists of an array of clusters, in which each cluster contains a numeric limit value, along with the corresponding color to display for that value. When you specify the color table in this manner, you can specify an upper out-of-range color using the **Z Scale Info: High Color** attribute, and a lower out-of-range color using the **Z Scale Info: Low Color**. The total number of colors is limited to 254 colors, with the lower and upper out of range colors bringing the total to 256 colors. If you specify more colors, then the 254 color table is created by interpolating between the specified colors.

Another way to set the colors programmatically is to specify a color table using the **Color Table** attribute. With this method you can specify an array of up to 256 colors. Data passed to the chart is mapped to indices in this color table based upon the color scale. If the color scale ranges from 0 to 100, a value of 0 in the data is mapped to index 1, and a value of 100 is mapped to index 254, with interior values interpolated between 1 and 254. Anything below 0 is mapped to the out of range below color (index 0), and anything above 100 is mapped to the out of range above color (index 255).

Note: Remember that the colors you want your intensity chart (or graph) to display are limited to the exact colors and number of colors that your video card has. You are also limited by the number of colors that LabVIEW allocates for your display.

Intensity Graph

The intensity graph is essentially the same as the intensity chart, except it does not retain previous data.

Each time it is passed new data, the new data replaces old data as it arrives.

For an example of an intensity graph, see `examples\graphs\intgraph.llb`.

Intensity Graph Data Type

The intensity graph accepts two-dimensional arrays of numbers, where each number is mapped by the chart to a color.

Rows of the data you pass in are displayed as new columns on the chart. If you want rows to appear as rows, use the **Transpose Array** option from the chart pop-up menu.

Intensity Graph Options

The intensity graph works much like the intensity chart, except it does not have the chart update modes. Because each update replaces the previous data, it does not have a scrollbar, and it does not have history options.

The intensity graph can have cursors like other graphs. Each cursor displays the X, Y, and Z values for a specified point on the graph. See Graph Cursors for more information on manipulating the graphs cursors.

You set the color mapping in the same way that you set it for the intensity chart.

Creating VIs

This topic discusses basic features you need to be familiar with in order to create or use VIs, including information about the front panel and block diagram windows, LabVIEW palettes and menus. It also discusses basic tasks you need to learn such as how to create objects, change tools, get help, and how to open, run, and save VIs.

Front Panel and Block Diagram
LabVIEW Menus
Creating Objects
Object Pop-Up Menus
Quick Access to Controls and Functions
LabVIEW Tools
Saving VIs
Running VIs
Stopping VIs
Getting Help

Front Panel and Block Diagram

Each VI has two separate but related windows: the front panel and the block diagram. You can switch between windows with the **Show Panel**/**Show Diagram** command in the **Windows** menu. Using the **Tile** commands, also in the **Windows** menu, you can position the front panel and block diagram windows side-by-side (next to each other), or up-and-down (one at the top of your screen, and one at the bottom of your screen).

If you have multiple VIs open simultaneously, only one is the active VI. This is the VI whose front panel or block diagram is frontmost or currently selected. All open front panels and block diagrams are listed at the bottom of the **Windows** menu, and the active front panel or block diagram has a checkmark beside its name.

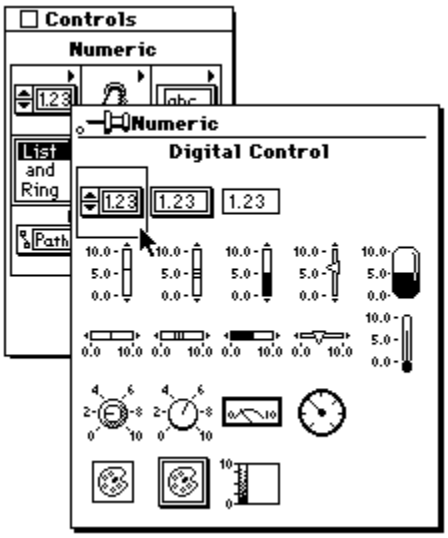
LabVIEW Menus

LabVIEW uses menus extensively. The *menu bar* at the top of a VI window (or at the top of the Macintosh screen) contains several *pull-down menus*. When you click on a menu bar item, a menu appears below the bar. The pull-down menus contain items common to other applications, such as **Open**, **Save**, **Copy**, and **Paste**, and many others particular to LabVIEW. Some menus also list shortcut key combinations. Click on the graphic below to explore the menus.

The LabVIEW menu you will use most often is the object *pop-up menu*. Virtually every LabVIEW object, as well as empty front panel and block diagram space, has a pop-up menu of options and commands. Instructions throughout this *LabVIEW Online Reference* suggest that you select a command or option from an object pop-up menu. To access an object's pop-up menu (or to "pop up on a menu"), put the cursor on that object and click the right mouse button (on the Macintosh, <command>-click the mouse button).

Creating Objects

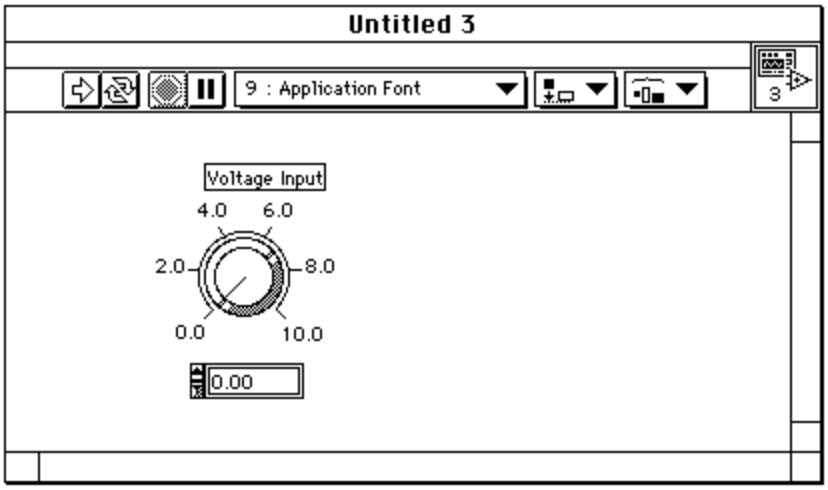
You create objects on the front panel and block diagram by selecting them from the floating **Controls** and **Functions** palettes. For example, if you want to create a knob on a front panel, you select it from the **Numeric** palette of the **Controls** palette as shown in the following illustration.



As you move the selection arrow over an object on the palette, the name of the object appears at the top of the palette. In the preceding illustration, the digital control is selected. At this point, if you click and move the mouse over the front panel, a digital control appears wherever you release the mouse, and a corresponding terminal appears in the block diagram. You can resize an object at the same time you create it by clicking and dragging as you place the object.

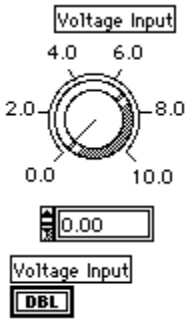
When you create front panel objects, they appear with a label rectangle ready for you to enter the name of the new object. If you want to give the object a name, enter the name on the keyboard. When you are finished entering the name, end text entry by pressing the <Enter> key on the numeric keypad. If you do not have a keypad, you can press the <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key on the keyboard, click on the enter button in the toolbar, or click outside the label.

The following illustration shows an example of the result of this action.



When you create an object on the front panel, a corresponding terminal is created on the diagram. You use this terminal if you want to read data from a control or send data to an indicator.

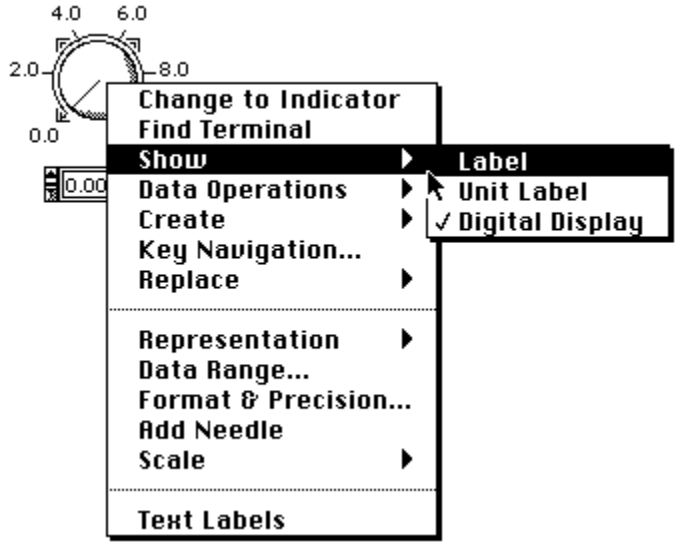
If you select **Windows»Show Diagram**, you can see the corresponding block diagram for the front panel. The block diagram contains terminals for all front panel controls and indicators.



Object Pop-Up Menus

All LabVIEW objects have associated pop-up menus. You pop up on an object by clicking on the object with the right mouse button (on the Macintosh, holding down the <command> key while you click on the object with the mouse). By selecting options from the resulting menu, you can access options related to that object and change its look or behavior.

For example, if you do not enter text into the label of a control when you first create it, the label disappears when you click elsewhere. You can show the label again by popping up on the control and selecting **Show»Label**.



Quick Access to Controls and Functions

If you need several functions from the same palette, you may want to keep a palette open. To keep a palette open, select the push-pin in the top left corner of the palette. Once you've pinned a window open, it has a titlebar so you can move it around easily. If you save the VI, the next time you launch LabVIEW it will open the palettes in the same locations that you last left them.

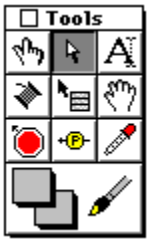
If you pop up in empty space in a front panel or block diagram window, a temporary **Controls** or **Functions** palette appears. If you are using a small monitor, you may want to close the floating **Controls** and **Functions** palettes and create objects with these pop-up palettes instead.

LabVIEW Tools

A *tool* is a special operating mode of the mouse cursor. You use tools to perform specific functions.

Many of LabVIEW's tools are contained in the floating **Tools** palette shown in the following illustration.

You can move this palette anywhere you want, or you can close it temporarily by clicking on the close box. Once closed, you can access it again by selecting **Windows»Show Tools Palette**. You can also bring up a temporary version of the **Tools** palette at the location of your cursor by clicking while pressing <Ctrl-Shift> (Windows); <command-Shift> (Macintosh); <meta-Shift> (Sun); or <Alt-Shift> (HP-UX). You can click on one of the tool icons in the palette below for a tool description and definition.



You change from one tool to another by doing any of the following while in edit mode:

- Click on the tool you want in the **Tools** palette.
- Use the <Tab> key to move through the most commonly used tools in sequence.
- Press the spacebar to toggle between the Operating tool and the Positioning tool when the front panel is active, and between the Wiring tool and the Positioning tool when the block diagram is active.

Saving VIs

This topic includes the following subtopics:

Saving VIs as Individual Files
Saving VIs in VI Libraries (.LLBs)

Saving VIs as Individual Files

Five options in the **File** menu concern saving your VIs as individual files.

Select the **Save** option to save a new VI, choose a name for the VI, and specify its destination in the disk hierarchy. Also use this option to save changes to an existing VI in a location previously specified.

If you want to save a VI with a new name, you can use **Save As…**, **Save a Copy As…**, or **Save with Options…** from the **File** menu.

When you select the **Save As...** option, LabVIEW saves a copy of the VI in memory to disk with the name you specify. After the save is finished, the VI in memory points to the new version. In addition, all callers to the old VI that are in memory now refer to the new VI. If you enter a new name for the VI, LabVIEW does not overwrite or delete the disk version of the original VI.

When you select the **Save A Copy As...** option, LabVIEW saves a copy of the VI in memory to disk with the name you specify. This does not affect the name of the VI in memory.

Save with Options... brings up a dialog box in which you can choose to save an entire VI hierarchy to disk, optionally saving VIs without their block diagrams. This option is useful when you are distributing VIs or making backup copies. See the Using Save with Options topic for instructions on how to use this option.

Caution: You cannot edit a VI after you save it without a block diagram. Always make a copy of the original VI.

You can use the **File»Revert...** option to return to the last saved version of the VI you are working on. A dialog box appears to confirm whether to discard all changes made to the VI.

VIs that have been modified since they were last saved are marked with an asterisk in their titlebars and in the list of open VIs displayed in the **Windows** menu. When you save a VI, the asterisk disappears until you make a new change.

See the Backing Up Your Files topic for information on saving backup copies.

Caution: **Do not save your VIs in the `vi.lib` directory. This directory is updated by National Instruments as needed during new version releases of LabVIEW. Placing VIs in vi.lib risks creating a conflict during future installations. If you want your VIs to show up in the Functions palette, see Customizing the Controls and Functions .**

Saving VIs in VI Libraries (.LLBs)

You can group several VIs together and save them as a VI library. However, unless you have a good reason for saving them as libraries, it is preferable to save them as individual files, organized in directories. Read the following lists before deciding to save VIs in a library.

Reasons for Saving Vis as Libraries
Creating a VI Library
Saving in an Existing VI Library
Editing the Contents of Libraries

Reasons for Saving VIs as Libraries

- If you are using Microsoft Windows 3.1 or plan to transfer your files to Windows 3.1, saving VIs as libraries allows you to use up to 255 characters to name your files. Other operating systems have long filename support (31 characters on the Macintosh, 255 characters on Windows 95, Windows NT, and UNIX systems.) "
- If you will be transferring VIs to other platforms, transferring a VI library is easier than transferring multiple individual VIs.
- If disk space is an important issue, you may prefer to save your files within VI libraries, because they are compressed, slightly reducing disk space requirements for VIs.

Reasons for saving VIs as individual files:

- If you store your VIs as individual files, you can use the file system to manage them (copying, moving, renaming, backing up, managing source code).
- You cannot have hierarchy within a VI library--VI libraries cannot contain subdirectories or folders.
- Loading and saving files are faster from the file system than from VI libraries. Less disk space is required for temporary files during the load and save processes.
- Storing VIs and controls in individual files is more robust than storing your entire project in the same file.

Note that many of the VIs shipped with LabVIEW are shipped in VI libraries so that they are stored in consistent locations on all platforms.

Creating a VI Library

To create a VI library, take the following steps:

1. Select **Save**, **Save as...**, or **Save a Copy As...** from the **File** menu.
2. ***(Macintosh)*** If LabVIEW is configured through **Preferences» Miscellaneous** to use native file dialog boxes, click on Use LLBs in the dialog box that appears.
3. In the dialog box that appears after step 1 or step 2, click on the **New...** button.

4. In the dialog box that appears, enter the name of the new library and click on the **VI Library** button. LabVIEW will add a `.llb` extension to the name of the library automatically, if you do not.
5. A dialog box appears, ready for you to name your VI if you wish, and save it in the new library.

Saving in an Existing VI Library

Once you have created a VI library, you can save new VIs in it much as if it were a directory. After selecting one of the Save options from the **File** menu, the name of your library file, with the `.llb` extension, appears in the file dialog box. (Macintosh users using the native file dialog box need to press the **Use LLBs** button first, before being able to select a library.) When you double-click the name of the library file or press **Open**, LabVIEW's dialog box lets you save the file in the library.

Editing the Contents of Libraries

You can remove a file from a VI library using the **File»Edit VI Library...** option. The Edit VI Library dialog box that appears also lets you mark files as **Top Level**, which has two uses. First, when you create an application using the application builder libraries, the **Top Level** setting indicates which VIs are to open automatically when you run the application. Second, if you double-click on a specific library from the file system on the Macintosh or Windows, or if you launch LabVIEW with a library name specified on a command line under Windows or UNIX, LabVIEW opens all top level VIs automatically.

For more complex editing operations, you can use the `llbedit.vi` in the `examples\llbedit` directory. This directory also contains several subVIs that can help you create your own tools for editing LLBs.

Running VIs

 You can run a VI by selecting **Operate»Run** or by clicking on the run button. LabVIEW compiles the VI if necessary.

 While the VI is executing, the run button changes appearance. If the VI is running at its top level (meaning it has no callers; it is not a subVI), the run button changes to look like the illustration shown to the left.

 If the VI is executing as a subVI, the run button changes to look like the illustration shown to the left.

 Pressing this button runs the VI over and over, until you stop or pause execution.

 Pressing this button aborts execution of the top level VI. If a VI is used by more than one running top level VI, the button is grayed out.

 Pressing this button pauses execution. You can press it again at any time to continue execution.

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

If your VI runs but does not perform correctly, see the Debugging VIs topic.

Stopping VIs

Normally, you should let a VI run until it completes. However, if you need to halt execution immediately, click on the stop button or select the **Operate»Stop**. The **Stop** command aborts the top level VI at the earliest opportunity. The halted VI most likely did not complete its task, and you cannot rely on any data it produces. Although LabVIEW closes files open at the time and halts any data acquisition that may be in progress, you should avoid designing VIs that rely on the **Stop** option. If you create a VI that executes indefinitely within a While Loop until stopped by the operator, for example, wire the conditional terminal of the loop with a Boolean switch on the front panel.

If you want to prevent an operator from inadvertently aborting your VI by clicking on the stop button, hide it by deselecting the **VI Setup...»Window Options»Show Abort Button** option from the icon pane pop-up menu on the front panel of the VI. See Creating Pop-Up Panels and Setting Window Features for information on setting window options.

Getting Help

The LabVIEW Help window offers help information for functions, constants, subVIs, controls and indicators, and dialog box options. To display the window, choose **Show Help** from the **Help** menu or press <Ctrl-h> (Windows); <command-h> (Macintosh); <meta-h> (Sun); or <Alt-h> (HP-UX). If your keyboard has a <Help> key, you can press that key instead. Move your cursor onto the icon of a function, a subVI node, or a VI icon (including the icon of the VI you have open, shown at the top right of the VI window) to see the help information.

Selecting **Help»Lock Help** or clicking on the lock icon at the bottom of the window locks the current contents of the Help window. Once you have locked it, moving over another function or icon does not change the display in the Help window. Select **Lock Help** or click on the lock icon again to turn this option off.

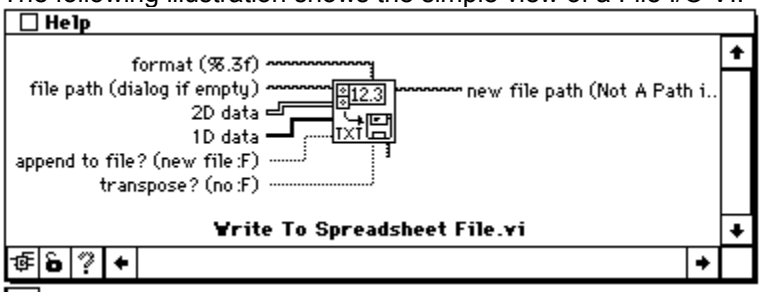
For more information on the Help window, see the following subtopics:

Block Diagram Help Views
Front Panel Help
Online Reference
Creating Your Own Help Files

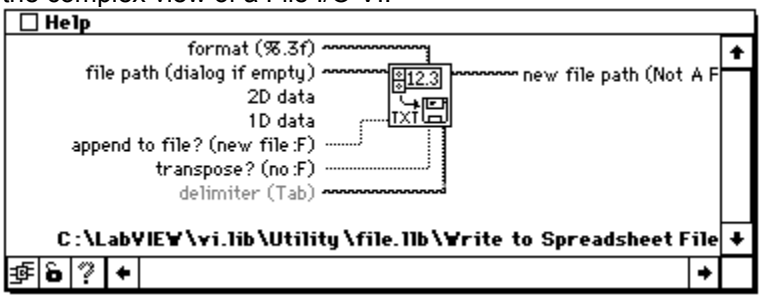
Block Diagram Help Views

You can display the Help window as either a simple or a complex view by pressing the second button at the bottom of the Help window or by toggling the **Help»Simple Help** option.

The simple view emphasizes the important connections and de-emphasizes the other connections. In this view, labels of required connections are in bold text and recommended connections are in plain text. Wire stubs appear in the place of the optional inputs and outputs that are not displayed, to inform you that additional connections exist, and that the complex view of the window will display them. The following illustration shows the simple view of a File I/O VI.



In the complex view, the Help window shows all inputs with wires pointing to the left, and outputs with wires pointing to the right. Labels of optional inputs are in gray text. The following illustration shows the complex view of a File I/O VI.



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

The terminal names for subVI nodes are the labels of the corresponding front panel controls and indicators. The default value of a subVI does not automatically appear in the wiring diagram. It is a good idea, however, to include the default value in parentheses in the name of controls when you create a subVI.
When you place the Wiring tool over a wire, the Help window displays the data type carried by that wire. Also, when you move the Wiring tool over the VI icon, as you pass over the different areas of the icon, the corresponding connector terminal is highlighted in the Help window.

Front Panel Help

When you move the cursor over a control or indicator, the Help window displays the description for that particular control or indicator. It is a good idea to enter descriptions for all controls and indicators when you create a VI. See Creating Descriptions for information.

If you hold the cursor on a VI icon in the top right of a front panel for a few moments, the Help window displays help for that VI.

Online Reference

You are currently using LabVIEW's extensive *Online Reference*, which you can access by selecting **Help»Online Reference**. In addition, for most block diagram objects you can select **Online Reference** from the object's pop-up menu to access the online description of the object. Another way of accessing this information is to press the button shown to the left, which is located at the bottom of LabVIEW's Help window.
For information on creating your own online reference files, see the Creating Your Own Help Files topic.

Operating Tool

Operating tool--Places **Controls** and **Functions** palette objects on the front panel and block diagram.

Positioning Tool

Positioning tool--Positions, resizes, and selects objects.

Labeling Tool

Labeling tool--Edits text and creates free labels.

Wiring Tool

Wiring tool--Wires objects together in the block diagram.

Object Pop Up Menu Tool

Object pop-up menu tool--Pops up on an objects menu.

Scroll Tool

Scroll tool--Scrolls through the window without using the scroll bars.

Breakpoint Tool

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

Probe Tool

Probe tool--Creates probes on wires.

Color Copy Tool

Color Copy tool--Copies colors for pasting with the Color tool.

Color Tool

Color tool--Sets foreground and background colors.

Editing VIs

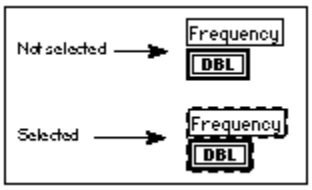
This topic discusses editing techniques for the front panel and the block diagram, under the following subtopics:

Selecting Objects
Moving Objects
Dragging and Dropping (Windows, Macintosh) Duplicating Objects
Deleting Objects
Labeling Objects
Resizing Objects
Creating Descriptions
Coloring Objects
Aligning and Distributing Objects
Moving Objects to Front and BackWhen you are familiar with these techniques, see the subtopics in the Front Panel Reference for information on building front panels, or the subtopics in the Block Diagram Reference for help in building block diagrams.

Selecting Objects

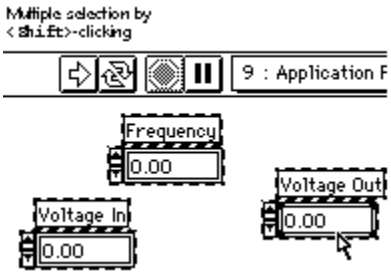
For many editing operations, such as moving, copying, and deleting, you must *select* an object.

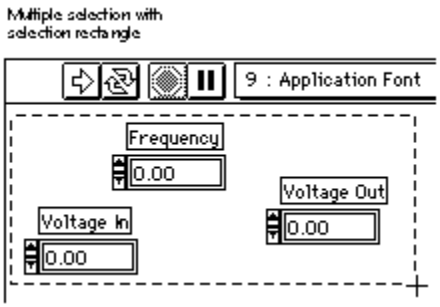
To select an object, click the mouse button while the Positioning tool, shown at the left, is on the object. When you select an object, LabVIEW surrounds it with a moving dashed outline called a *marquee*.



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

Another way to select single or multiple objects is to drag a selection rectangle around the object(s), as shown in the following illustration.





Click in an open area with the Positioning tool and drag diagonally until all the objects you want to select lie within or are touched by the selection rectangle that appears. When you release the mouse button, the selection rectangle disappears, and a marquee surrounds each selected object. You can then move, copy, or delete the selected objects.

Some objects must be completely enclosed by the rectangle to be selected. Others are selected when they are partially enclosed. By holding down the <Shift> key while dragging the selection rectangle, you can select additional objects or deselect an object enclosed by the rectangle.

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

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

Moving Objects

You can move an object by clicking on it with the Positioning tool and then dragging it to the desired location.

If you hold down the <Shift> key and then drag an object, LabVIEW restricts the direction of movement horizontally or vertically, depending on which direction you first move the object.

You can move selected objects in small, precise increments by pressing the appropriate arrow key on the keyboard once for each pixel you want the objects to move. Hold down an arrow key to repeat the action. Hold down the <Shift> key along with an arrow key to move the object more rapidly.

If you change your mind about moving an object while you are dragging, continue to drag until the cursor is outside all open windows and the dotted outline disappears, then release the mouse button. This cancels the move operation, and the object remains where it was. If your screen is cluttered, the menu bar is a convenient and safe place to release the mouse button when you want to cancel a move.

Dragging and Dropping (Windows, Macintosh)

On Windows and the Macintosh, you can drag VIs from the file system to a LabVIEW diagram to create a subVI call to that VI. This feature also works for custom controls, type definitions, and globals. In addition, you can drag text and pictures from other applications to copy them to front panels and block diagrams.

On the Macintosh, your system must have the Drag Manager in order to use external Drag and Drop. The Drag Manager is built into System 7.5 and later. It is also available for System 7.0 up to System 7.5 as an extension from Apple.

Under Windows 3.1, external drag support is limited to dragging VIs and controls from the File Manager. Under Windows 95 and Windows NT, which have 32-bit OLE support, you can drag text and pictures from OLE-supporting applications, in addition to dragging from the File Manager/Explorer.

You can use the following Drag and Drop capabilities internally (between various LabVIEW objects):

- Drag front panel controls and indicators to block diagrams to create block diagram constants and vice versa.
- Drag a subVI to a path control or constant to drop the full pathname of the VI inside the control or constant.
- You can use the following Drag and Drop capabilities externally (from other applications to LabVIEW objects):
- Drag a file into a path control or constant to drop its full pathname.
- Drag a datalog file into a Data Log File Refnum to create a cluster containing the data structure in the datalog file.
- Drag a graphics file into a Pict Ring, front panel, or block diagram to drop the picture it contains.
- Drag a VI file into the block diagram of a VI to drop it as a subVI.
- Drag a custom control file into the front panel to drop the custom control stored in that file.
- ***(Macintosh)*** Drag a Text Clipping or text selected from any application to drop that text into a string control or constant, front panel, block diagram, or label.
- ***(Macintosh)*** Drag an Image clipping or an image selected from another application to drop it into a picture ring, front panel, or block diagram.
- ***(Windows 95 and Windows NT)*** Drag Text from an OLE source to drop its text into a string control or constant, front panel, block diagram, or label.
- ***(Windows 95 and Windows NT)*** Drag Image from an OLE source to drop its picture into a picture ring, front panel, or block diagram.

Notice that if an object can be successfully dropped, the destination is highlighted.

For more information on Drag and Drop features, see your Windows or Macintosh system manual.

Duplicating Objects

There are three basic methods for duplicating a LabVIEW object--by copying and pasting, by cloning, and by dragging and dropping. In all cases, LabVIEW creates a new copy of the object complete with its associated elements, such as the terminal belonging to a front panel control or the front panel control belonging to a terminal, the label, Attribute Nodes, and locals. Locals and Attribute Nodes referring to original controls are copied and switched so that they refer to the copied control.

When you clone or copy objects, the copies are labeled with the same name as the original with the word copy appended (or copy 1, copy 2, etc. for copies of copies).

To copy and paste within a VI or between VIs, select the object with the Positioning tool and choose **Edit»Cut** or **Edit»Copy**. Then click where you want to place the duplicate and choose **Edit»Paste**. You can duplicate several objects at the same time by dragging a selection marquee around the objects before duplicating them. You can also copy text or pictures from other applications and paste them into LabVIEW.

To clone an object, click the Positioning tool over it while pressing the `<Ctrl>` (Windows); `<option>` (Macintosh); `<meta>` (Sun); or `<Alt>` (HP-UX) key, and drag the copy to its new location. In UNIX, you can also clone by clicking and dragging the middle mouse button without modifier keys.

On the Macintosh and in Windows, you can use Drag and Drop capabilities to copy objects between VIs or from other applications. To drag and drop an object, select it with the Positioning tool and drag it to the other panel or diagram to which you want to copy it. For more information, see (Windows, Macintosh) Dragging and Dropping VIs, Pictures, and Text to LabVIEW.

Instead of duplicating data types for existing front panel controls, you can copy, clone, or drag and drop them to any diagram to make a corresponding constant. In the same way, you can drag user-defined

constants from the block diagram to the front panel to create controls.

Deleting Objects

To delete an object, select the object and choose **Edit»Clear** or press the <Backspace> (Windows and HP-UX) key or the <Delete> key (Macintosh and Sun). You can only delete block diagram terminals by deleting the corresponding front panel controls or indicators.

Although you can delete most objects, you cannot delete parts of a control or indicator such as labels or digital displays. However, you can hide these components by deselecting **Show»Label** or **Show»Digital Display** from the objects pop-up menu.

If you delete a structure such as a While Loop, you normally delete its contents as well. If you want to delete only the structure itself but preserve its contents, pop up on the edge of the structure and select **Remove While Loop** (or other structure name). LabVIEW deletes the structure but not its contents and automatically reconnects any wires that crossed the border of the structure.

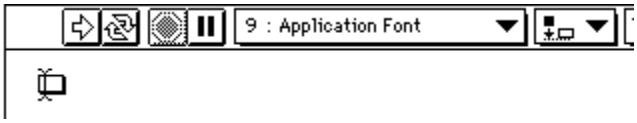
Labeling Objects

Labels are blocks of text that annotate components of front panels and block diagrams. There are two kinds of labels--*owned* labels and *free* labels. Owned labels belong to and move with a particular object and annotate that object only. You can hide these labels but you cannot copy or delete them independently of their owners. Free labels are not attached to any object, and you can create, move, or dispose of them independently. Use them to annotate your front panels and block diagrams. You use the Labeling tool, whose cursor is the I-beam and box cursor, to create free labels or to edit either type of label.

Creating Labels
Changing Font, Style, Size, and Color of Text

Creating Labels

To create a free label, select the Labeling tool from the **Tools** palette and click anywhere in empty space.



A small box appears with a text cursor at the left margin ready to accept typed input. Type the text you want to appear in the label. As shown in the following illustration, the *enter* button appears on the toolbar to remind you to end text entry by clicking on the enter button.

You can also end text entry by pressing the <Enter> key on the numeric keypad. If you do not have a keypad, you can press the <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key on the keyboard, or click outside the label. If you do not type any text in the label, the label disappears as soon as you click somewhere else.

Note: If you place a label or any other object over (partially covering) a control or indicator, it slows down screen updates and could make the control or indicator flicker. To avoid this problem, do not overlap a front panel object with a label or other object.

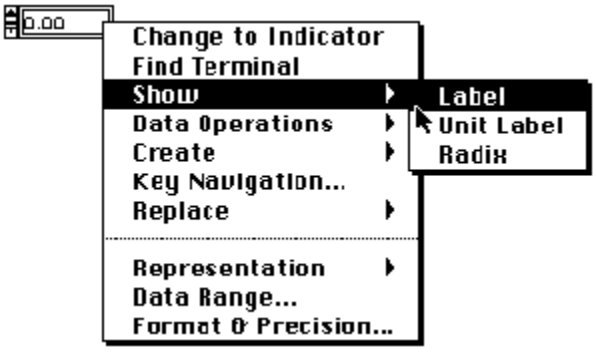
You can copy the text of a label by selecting it with the Labeling tool. Double-click or triple-click on the text

with the Labeling tool or drag the Labeling tool across the text to highlight it. When the text is selected, choose **Edit»Copy** to copy the text onto the Clipboard. Now you can highlight the text of a second label. Select **Edit»Paste** to replace the highlighted text in the second label with the text from the Clipboard. To create a new label with the text from the Clipboard, click on the screen with the Labeling tool where you want the new label positioned. Then, select **Edit»Paste**.

When you create a control or indicator on the front panel, a blank, owned label accompanies it, waiting for you to type the name of the new control or indicator. The label disappears if you do not enter text into it before clicking elsewhere with the mouse.

Structures and functions come with a default label, which is hidden until you show it. You can edit this label as desired.

To display a hidden name label, pop-up on the object and select **Show»Label** from the pop-up menu, as shown in the following illustration.

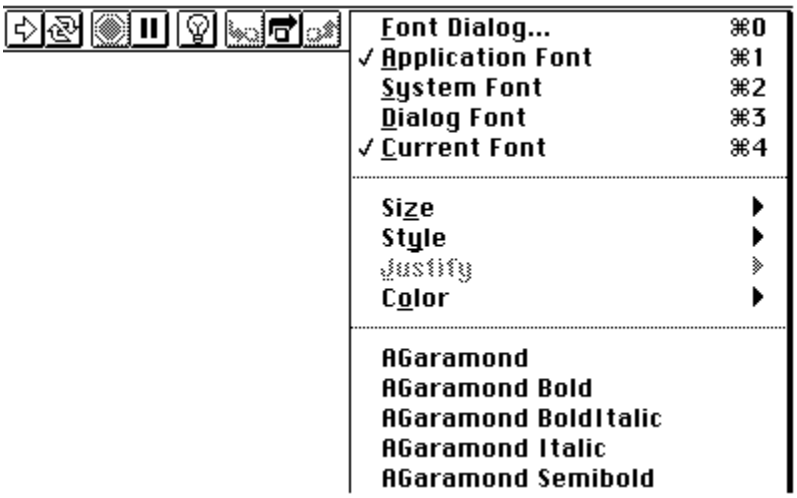


The label appears, waiting for typed input. The label disappears if you do not enter text into it before clicking outside the label.

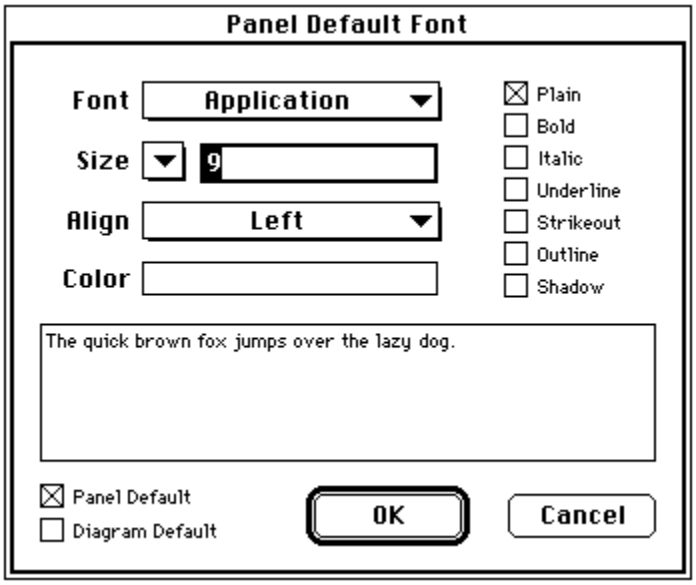
Note: On the block diagram, subVIs do not have labels that you can edit. The label of a subVI always shows its name. Function labels, on the other hand, can be edited to reflect the use of the function in the block diagram. For example, you can use the label of an Add function to document what quantities are being added, or why they are being added at that point in the block diagram. See the note in Creating Descriptions for related information.

Changing Font, Style, Size, and Color of Text

You can change text attributes in LabVIEW using the options on the Font ring on the toolbar, as shown in the illustration that follows. If you select objects or text and make a selection from this menu, the changes apply to everything selected. If nothing is selected, the changes apply to the default font, meaning that labels you create from then on will use the new default font. Changing the default font does not change the font of existing labels; it only affects labels created from that point on.



If you select **Font Dialog...** while a front panel is active, the dialog box shown in the following illustration appears. If a block diagram is active instead, the dialog box differs only in the selection of **Diagram Default** instead of **Panel Default** in the checkboxes at the bottom of the dialog box.



With either the **Panel Default** or **Diagram Default** checkbox selected, the other selections made in this dialog box will be used with new labels on the front panel or block diagram.

Notice the word **Application** showing in the Font ring, which also contains the **System**, **Dialog**, and **Current** options. The last option in the ring, **Current Font**, refers to the last font style selected.

LabVIEW uses the Application, System, and Dialog fonts for specific portions of its interface. These fonts are predefined by LabVIEW so that they map best between platforms. When you take a VI that contains one of three fonts to another platform, LabVIEW ensures that the font maps to something reasonable.

The predefined fonts are as follows:

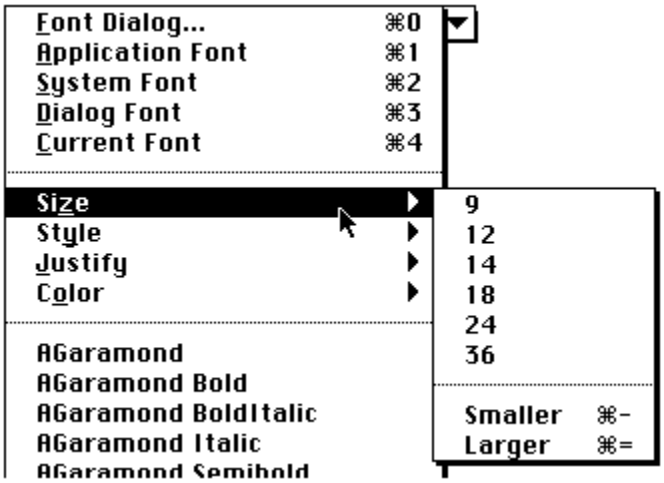
- The Application font is the LabVIEW d*efault* font. It is used for the **Controls** palette, the **Functions** palette, and text in new controls.
- The System font is the font LabVIEW uses for menus.
- The Dialog font is the font LabVIEW uses for text in dialog boxes.

If you click the Panel Default and Diagram Default checkboxes, the selected font becomes the current

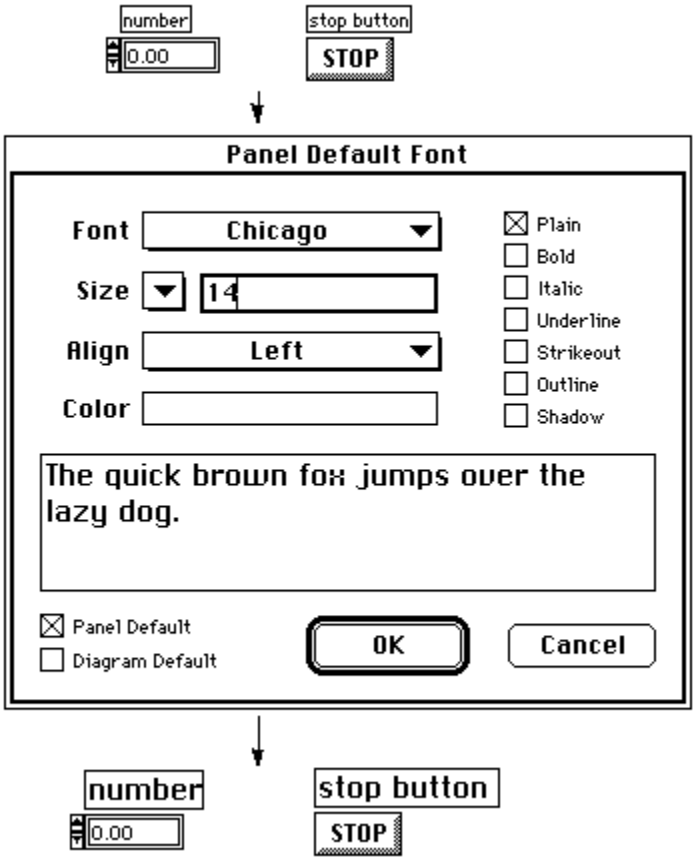
font for the front panel, the block diagram, or both. The current font is used on new labels. The checkboxes allow you to set different fonts for the front panel and block diagram. For example, you could have a small font on the block diagram and a large one on the front panel.

For information on portability issues regarding the three types of fonts, see the Resolution and Font Differences topic. For information on changing the defaults for the three categories of fonts, see Font Preferences.

The Font ring also has **Size**, **Style**, **Justify**, and **Color** options, as shown in the following illustration.

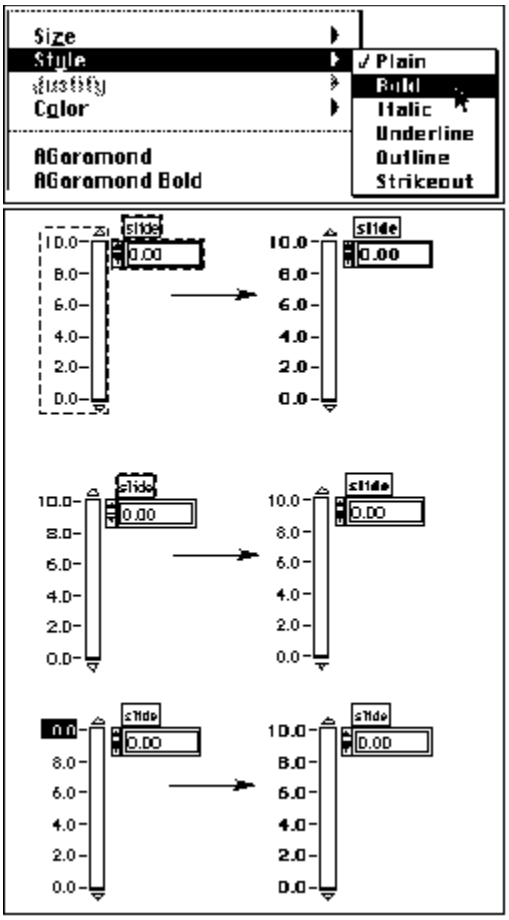


Font selections made from any of these submenus apply to objects you select. For example, if you select a new font while you have a knob and a graph selected, the labels, scales, and digital displays all change to the new font. The following illustration shows a numeric and a Boolean control being changed from the current font to the System font.



Notice that as many font attributes as possible are preserved when you make a change. If you change several objects to Courier font, the objects retain their size and styles if possible. In the same way, changing the size of multiple text selections does not cause the selections to have the same font. These rules do not apply if you select one of the predefined fonts, the current font, or when you use the Font Dialog box, which changes the selected objects to the selected font and set of attributes.

When working with objects like slides, which have multiple pieces of text, remember that text selections affect the objects or text currently selected. For example, if you select the entire slide while selecting **Bold**, the scale, digital display, and label all change to a bold font. If you select only the label while selecting **Bold**, only the label changes to bold. If you select text from a scale marker while selecting **Bold**, all the markers change to bold. These three types of changes are shown in the following illustration.

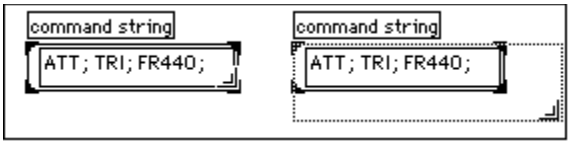


Resizing Objects

You can change the size of most objects. When you move the Positioning tool over a resizable object, resizing handles appear at the corners of the object, as shown in the following illustration.

Resizing tool

When you pass the tool over a resizing handle, the cursor changes to the *Resizing tool*. Click and drag this cursor until the dashed border outlines the size you want, as shown in the following illustration.



To cancel a resizing operation, continue dragging the frame corner outside the window until the dotted frame disappears. Then release the mouse button. The object maintains its original size.

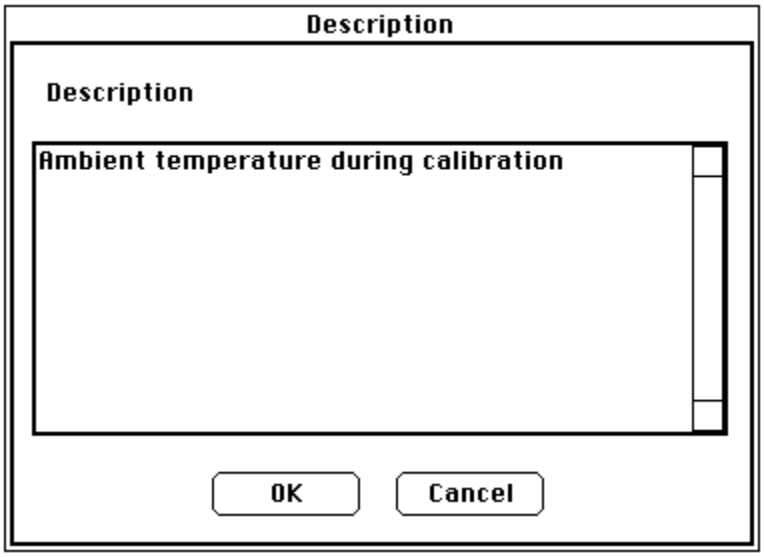
Some objects can change size only horizontally or vertically, or keep the same proportions when you resize it, such as a knob. The resizing cursor appears the same but the dotted resize outline moves in only one direction. To restrict the growth of any object vertically or horizontally, or to maintain the current proportions of the object, hold down the `<Shift>` key as you click and drag.

Resize labels as you would other objects, using the resizing handles. Labels normally *autosize*; that is, the box automatically resizes to contain the text you enter. Label text remains on one line unless you enter a carriage return or resize the label box. Choose **Size to Text** from the label pop-up menu to turn autosizing back on.

To add more working space to your panel or diagram, first <Ctrl-click> (Windows); (<option-click> (Macintosh); <meta-click> (Sun); <Alt-click> (HP-UX), and then drag out a region with the Positioning tool. You see a rectangle marked by a dotted line, which defines your new space. Objects on the front panel or block diagram move to accommodate the new space.

Creating Descriptions

If you want to enter a description of a LabVIEW object, such as a control or indicator, choose **Data Operations»Description...** from the objects pop-up menu. You must be in edit mode to edit a description. Enter the description in the dialog box, shown in the following illustration, and click **OK** to save it. LabVIEW displays this description whenever you subsequently choose **Description...** from the objects pop-up menu.



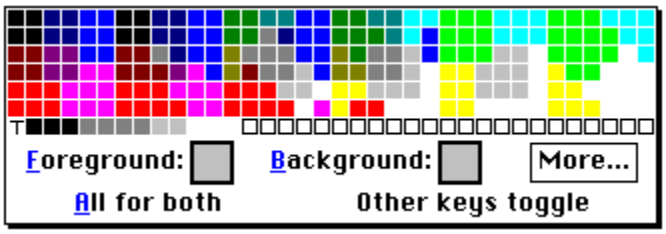
LabVIEW also displays the description of the object in the Help window when you place the cursor over the object. The best way to set up help for the VIs you create is to enter descriptions for all of your controls and indicators.

Note: You cannot edit subVI descriptions from the calling VI diagram. You can edit a VI description through the Get Info... selection of the File menu when the VIs front panel is open. Function description boxes are blank so that you can describe the action taking place at each occurrence of that function on the block diagram.

Coloring Objects

LabVIEW appears on the screen in black and white, shades of gray, or color depending on the capability of your monitor. You can change the color of many LabVIEW objects, but not all of them. For example, block diagram terminals of front panel objects and wires use color codes for the type and representation of data they carry, so you cannot change them. You cannot change colors in black-and-white mode.


Color tool To change the color of an object or the background of a window, pop up on it with the Color tool, as shown at the left. The following palette appears in color.



As you move through the palette while pressing the mouse button, the object or background you are coloring redraws with the color the cursor is currently touching. This gives you a preview of the object in the new color. If you release the mouse button on a color, the object retains the selected color. To cancel the coloring operation, move the cursor out of the palette before releasing the mouse button.

If the selected object can be made transparent, a T appears in one of the color rectangles. If you select that box, LabVIEW makes the object transparent. With this feature, you can layer objects. For example, you can place invisible controls on top of indicators, or you can create numeric controls without the standard three-dimensional container. Transparency affects only the appearance of an object. The object responds to mouse and key operations as usual.
Some objects have both a foreground and a background that you can color separately. The foreground color of a knob, for example, is the main dial area, and the background color is the base color of the raised edge. The display at the bottom of the color selection box indicates whether you are currently coloring the foreground, the background, or both. A black border around a square indicates that you have selected that square. In the default setting, both the foreground and the background are selected.

To change between foreground and background, you can press <f> for *foreground* and <b> for *background*. Pressing <a> for *all* selects both foreground and background. Pressing any other key also toggles the selection between foreground and background. The selection does not toggle until you move the Color tool.

Selecting the **More...** option from the Color palette calls up a dialog box with which you can customize the colors. The More Colors dialog box is shown in the following illustration.

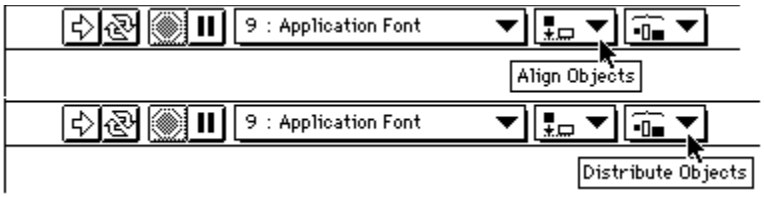
Each of the three color components, red, green, and blue, describes eight bits of a 24-bit color. Therefore, each component has a range of 0 to 255. To change the value of a color component, you can double-click in that colors display and enter the new value, or click on the appropriate arrow key to increment or decrement the current value. After entering the new value, press the <Enter> key on the numeric keypad to see the results in the color rectangle. To alter one of the base colors, click on the color rectangle and choose one of the selections. The component values for the selected color appear in each display.

The last color you select from the palette becomes the current color. Clicking on an object with the Color tool sets that object to the current color.

You can also copy the color of one object and transfer it to a second object without using the Color palette. Click with the Color Copy tool <Ctrl-click> (Windows); <command-click> (Macintosh); <meta-click> (Sun); or <Alt-click> (HP-UX) with the Color tool on the object whose color you want to duplicate. The Color tool appears as an eye dropper and takes on the color of the selected object. Now you can click on another object with the Color tool, and that object becomes the color you chose.

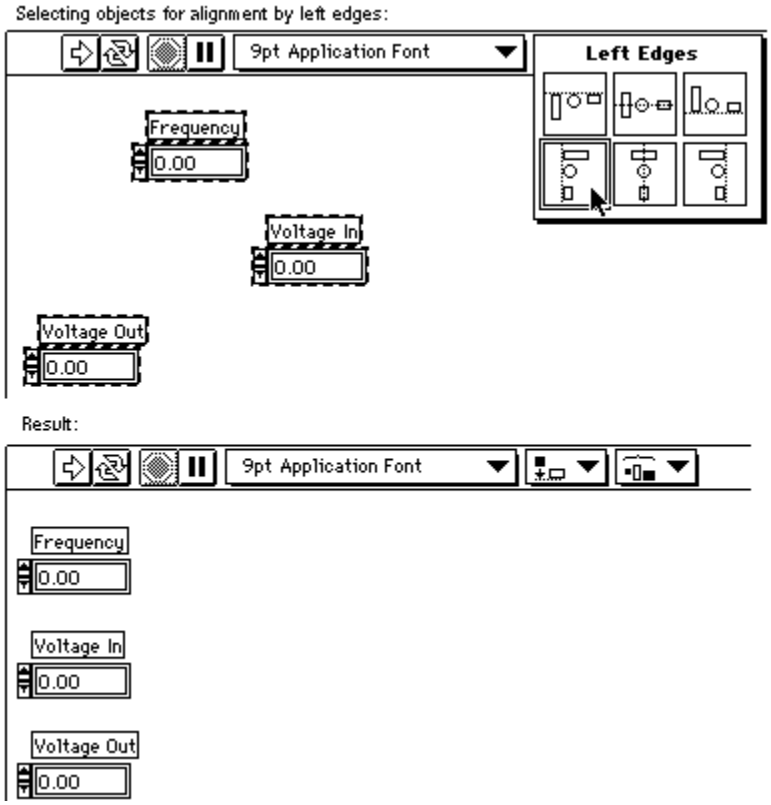
Aligning and Distributing Objects

LabVIEW has an automatic alignment mechanism, which is available through the Alignment and Distribution rings, which are accessed from the toolbar, as shown in the following illustration.



Aligning Objects

Select the objects you want to align and choose the axis along which you want to align them from the Alignment ring, as shown in the following illustration.



You can align an object along the vertical axis using its left, center, or right edge. You can also align an object along a horizontal axis using its top, center, or bottom edge. Your selection becomes the current alignment option, which is indicated by a dark border around the item in the palette.

- **Top Edges**--Aligns all top edges of the selected objects with the top-most object.
- **Vertical Centers**--Aligns the objects at the point halfway between the top-most and bottom-most objects.

- **Bottom Edges**--Aligns all bottom edges of the selected objects with the bottom-most object.
- **Left** Edges--Aligns all left edges of the selected objects with the left-most object.
- **Right Edges**--Aligns the edges of the selected objects with the right-most object.
- **Horizontal Centers**--Aligns the objects at the point halfway between the left-most and right-most objects.

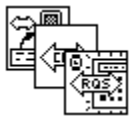
Distributing Objects

To space objects evenly, or distribute them, select the objects you want to distribute and choose how you want to distribute them from the Distribution ring. In addition to distributing selected objects by making their edges or centers equidistant, four options at the right side of the ring let you add or delete gaps between the objects, horizontally or vertically.

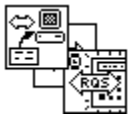
Moving Objects to Front and Back

You can place objects on top of other objects. LabVIEW has several commands in the **Edit** menu to move them relative to each other.

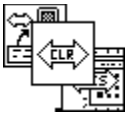
For example, assume you have three objects stacked on top of each other. Object 1 is on the bottom of the stack, and object 3 is on top.



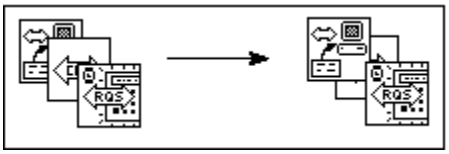
Move To Front moves the selected object in the stack to the top. If object 1 is selected, then object 1 moves to the top of the stack, with object 3 under it, and object 2 on the bottom.



If object 2 is then selected, **Move To Front** changes the order to object 2 on top, object 1 under it, and object 3 on the bottom.



Move Forward moves the selected object one position higher in the stack. So starting with the original order of object 1 on the bottom of the stack and object 3 on top, selecting this option for object 1 puts object 2 on the bottom, object 3 on the top, and object 1 in the middle.



Move To Back and **Move Backward** work similarly to **Move To Front** and **Move Forward** except that they move items down the stack rather than up.

Creating SubVIs

This topic discusses the hierarchical design of LabVIEW applications, under the following subtopics:

Importance of Hierarchical Design
Creating a SubVI from a VI
Creating a SubVI from a Selection
Hierarchy Window

Importance of Hierarchical Design

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

When creating an application, you start at the top-level VI and define the inputs and outputs for the application. Then you use other VIs as 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, you can group them into a lower level VI to maintain the simplicity of the block diagram. This modular approach makes applications easy to debug, understand, and maintain.

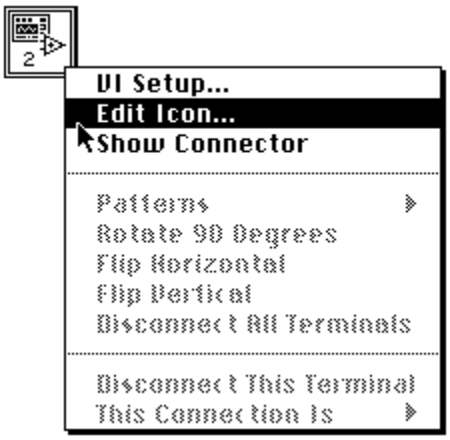
Creating a SubVI from a VI

The *icon* of a VI is its graphical symbol. The *connector* of a VI assigns controls and indicators to input and output terminals. If you want to call your VI from the block diagram of another VI, first create an icon and connector for it. This section explains how to create and edit a VI icon and connector, under the following subtopics:

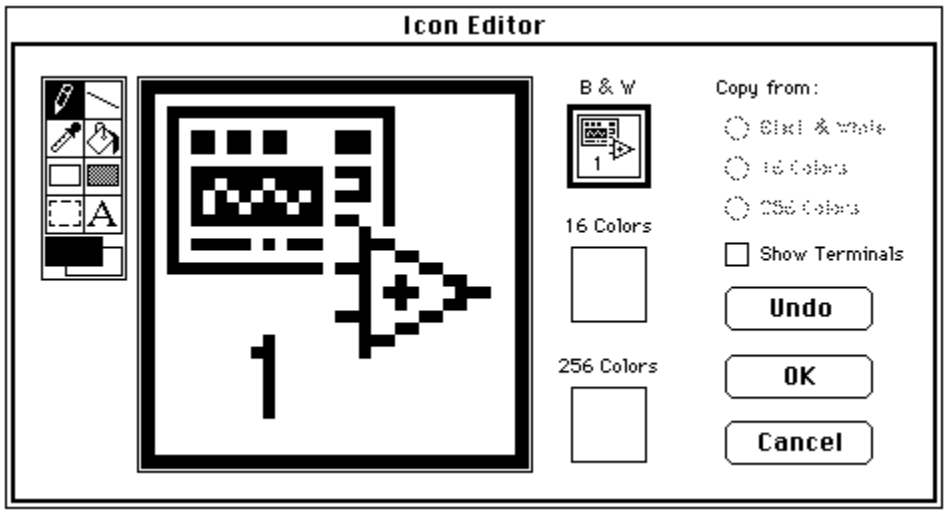
Creating the Icon
Defining the Connector Terminal Pattern
Selecting and Modifying Terminal Patterns
Assigning Terminals to Controls and Indicators
Required Connections for SubVIs
Deleting Connections
Confirming Connections

Creating the Icon

To create an icon, make sure you are in edit mode. Either double-click on the icon in the top-right corner of the front panel, or pop up on the icon and select **Edit Icon...**, as shown in the following illustration. Remember, you must be in edit mode in the front panel window to get this menu.



After you select **Edit Icon...**, the Icon Editor appears.



You use the tools to the left of the window, shown in the preceding illustration, to create the icon design in the fat-pixel editing area. The normal-size image of the icon appears in the appropriate box to the right of the editing area.

Depending on the type of monitor you are using, you can design a separate icon for display in 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. You can copy from a color icon to a black-and-white icon, and from black and white to color as well, by using the **Copy from** buttons at the right.

Note: It is best to always create a black-and-white icon. If you design a color icon, it does not show up in a palette of the Functions palette if you add it to the menus. It is not printed out and does not show up on a black-and-white monitor. LabVIEW uses the blank black-and-white icon in these cases.

The tool icons to the left of the editing area perform the following functions.

Pencil Tool

Draws and erases pixel by pixel. Use the <Shift> key to restrict drawing to horizontal and vertical lines.

Line Tool

Draws straight lines. Use the <Shift> key to restrict drawing to horizontal, vertical, and

diagonal lines.

Dropper Tool

Selects a color to be the foreground color from an element in the icon. Use the <Shift> key to select the background color with the dropper.

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. Use the <Shift> key to constrain the rectangle to be a square.

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. Use the <Shift> key to constrain the rectangle to a square.

Select Tool

Selects an area of the icon for moving, copying, or deleting. Double-click to select the entire icon. Double-click and then press the <Delete> key to erase the entire icon. Use the <Shift> key to constrain the rectangle to a square.

Text Tool

Enters text into the icon. Double-click on this tool icon to select a different font.

Foreground/Background Tool

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

Holding down the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key temporarily changes all of the tools except the select tool to the dropper.

The buttons to the right of the editing screen perform the following functions when you click on them.

Undo Cancels the last operation you performed in the Icon Editor .

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

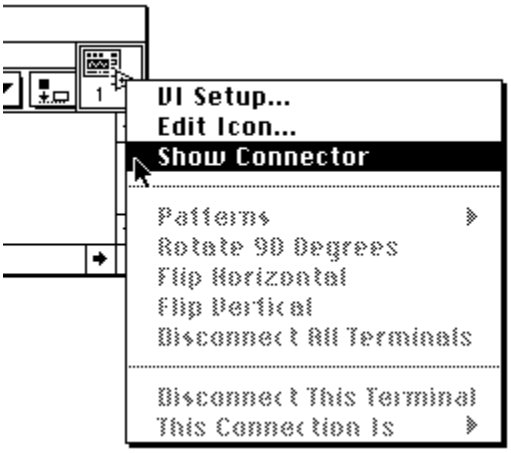
Cancel Returns to the front panel without saving any changes.

Note: You can cut, copy, and paste the entire icon using these options from the menu or pressing <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) plus x for cut, c for copy, and v for paste.

Defining the Connector Terminal Pattern

You send data to and receive data from a subVI through the terminals in its connector pane. You define connections by choosing the number of terminals you want for the VI and by assigning a front panel control or indicator to each of those terminals. Only the controls and indicators you use programmatically by wiring to the subVI require terminals on the connector pane.

If the connector for your VI is not already displayed in the upper right corner of the front panel, choose **Show Connector** from the icon pane pop-up menu, as shown in the following illustration. The block diagram does not have a connector pane.

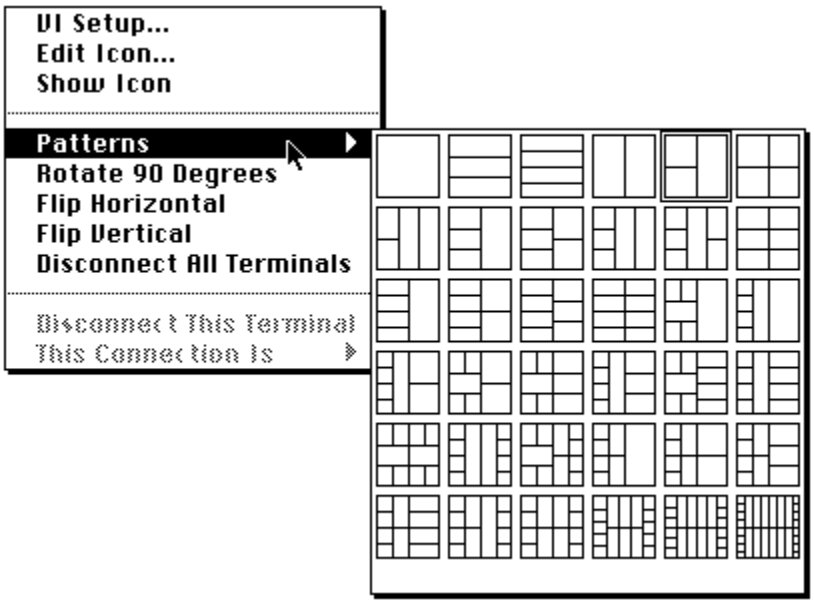


The connector replaces the icon in the upper right corner of the front panel. LabVIEW selects a terminal pattern for your VI with as many terminals on the left of the connector pane as controls on the front panel, and as many terminals on the right of the connector pane as indicators on the front panel.

Each of the rectangles on the connector represents a terminal area, and you can use them either for input to or output from the VI. If you want to use a different terminal pattern for your VI, you can select a different pattern.

Selecting and Modifying Terminal Patterns

To select a different terminal pattern for your VI, pop up on the connector and choose **Patterns** from the pop-up menu.



A solid border highlights the pattern currently associated with your icon, as shown in the previous illustration. To change the pattern, choose a new one. If you choose a new pattern, any assignment of controls and indicators to the terminals on the old connector pane is lost.

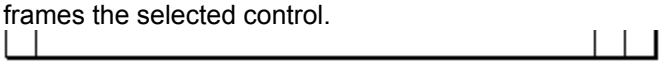
If you want to change the spatial arrangement of the connector terminal patterns, choose one of the following commands from the connector pane pop-up menu: **Flip Horizontal**, **Flip Vertical**, or **Rotate 90 Degrees**. LabVIEW disables these items if any terminal connections exist.

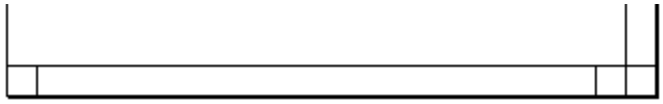
Assigning Terminals to Controls and Indicators

After you decide which terminal pattern to use for your connector, you must then assign front panel controls and indicators to the terminals. Follow these steps.

1. Click on a terminal of the connector. The tool automatically changes to the Wiring tool. The terminal turns black.

2. Click on the front panel control or indicator you want to assign to the selected terminal. A marquee frames the selected control.



3.  d click. The marquee disappears and the al has been assigned.

Note: Although you use the Wiring tool to assign terminals on the connector to front panel controls and indicators, no wires are drawn between the connector and these controls and indicators.

4. Repeat steps 1 and 2 for each control and indicator you want to connect.

You can also select the control or indicator first and then select the terminal. You can choose a pattern with more terminals than you need. You can leave some extra terminals unconnected if you anticipate making changes to the VI in the future that might require a new input or output. Having the extra connections available means that the new input or output does not affect other VIs that are already using this VI as a subVI. Unassigned terminals do not affect the operation of the VI. You can also have more front panel controls than terminals.

The connector pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector.

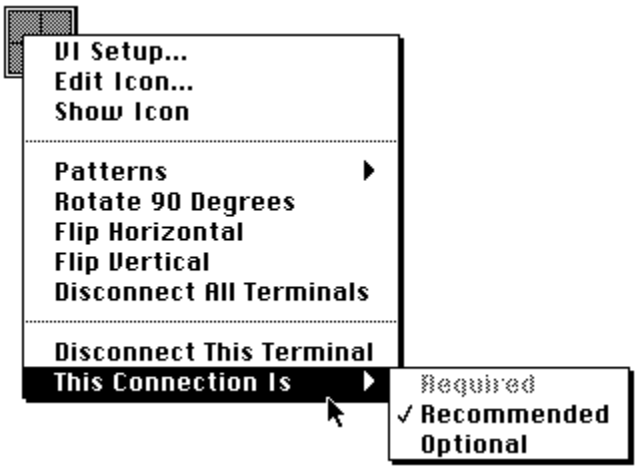
Required Connections for SubVIs

LabVIEW has a feature that can keep you from forgetting to wire subVI connections--indications of required, recommended, and optional connections in the connector pane and the same indications in the Help window.

Inputs and outputs of VIs that come in `vi.lib` have been pre-marked as required, recommended, or optional. Inputs and outputs of VIs that you create are set to recommended by default.

When an input is marked as required, you cannot run the VI as a subVI without wiring it correctly. When an input or output is marked as recommended, you can run the VI, but the Error List window will list a warning if warnings are enabled.

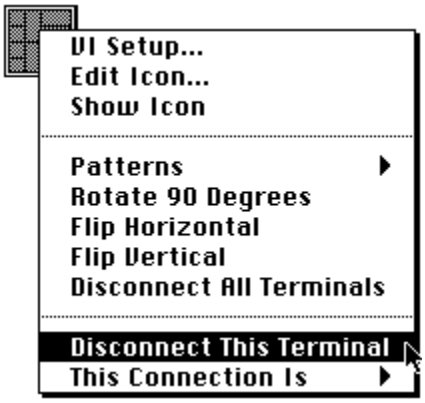
To see whether connections are required, recommended, or optional, or to mark them as one of these states, click on a terminal in the connector pane and select **This Connection Is**. A checkmark indicates its marking, as shown in the following illustration.



In the Help window, required connections appear in bold, recommended connections are in plain text, and optional connections are in grayed out text.

Deleting Connections

You can delete connections between terminals and their corresponding controls or indicators individually or all at once. To delete a particular connection, pop up on the terminal you want to disconnect on the connector and choose **Disconnect This Terminal** from the pop-up menu, as shown in the following illustration.



The terminal turns white, indicating that the selected connection no longer exists. To delete all connections on the connector, choose **Disconnect All Terminals** from the pop-up menu anywhere on the connector. LabVIEW also deletes all existing connections automatically if you select a new pattern from the **Patterns** palette. If you delete a control or indicator from the front panel, it is disconnected from its connector terminal, if one exists.

Confirming Connections

To see which control or indicator is assigned to a particular terminal, click on a control, indicator, or terminal with the Wiring tool when the connector pane is visible. LabVIEW selects the corresponding assigned object.

Creating a SubVI from a Selection

You can convert a portion of a VI into a subVI that can be called from another VI. You select a section of a VI, select **Edit»SubVI From Selection**, and the section becomes a subVI. Controls and indicators are automatically created for the new subVI, the subVI is automatically wired to the existing wires, and an

icon of the subVI replaces the selected section of the block diagram in your original VI.

Behaviors of The SubVI from a Selection Feature
Rules and Recommendations for Creating a SubVI from a Selection

Behaviors of the SubVI from a Selection Feature

Creating a subVI from a selection is the same as removing the selected objects and replacing them with a subVI, except for the following behaviors:

None of the front panel terminals included in the selection are removed from the caller VI. Instead, the front panel terminals are retained on the caller VI and are wired to the subVI.

All attributes included in the selection are retained on the caller VI and are wired to the subVI. The selected attributes are replaced by front panel terminals in the subVI which act as channels for transferring the attributes value, in or out of the subVI.

When a selection includes a local, it is replaced by a front panel terminal in the subVI. The local is retained on the caller VI and is wired to the subVI. When more than one instance of the same local is selected, the first instance becomes a front panel terminal and the rest remain as locals in the subVI. Only one instance of the local is retained on the caller VI; the rest of the selected instances are removed from the caller and moved to the subVI.

Because locals can either read or write from a front panel terminal, when instances of both read and write locals are selected, two front panel objects are created in the subVI--one to pass the value of the local into the subVI and another to pass the value out of the subVI. The front panel terminal created to represent the read locals has the suffix read added to its name, and the front panel terminal created to represent the write locals has the suffix write added to its name.

Note: Currently there is no Undo option to undo the creation of a subVI. However, selecting File»Revert can undo the creation of a subVI by reverting to the VI as it existed just before the last Save.

Rules and Recommendations for Creating a SubVI from a Selection

The ability to create a subVI from a selection can be a great convenience, but may not be as simple as it seems. Careful planning is still required to create a logical hierarchy of VIs. Moreover, the objects included in a selection must be carefully considered, in order to avoid situations where the functionality of a resulting VI would be adversely affected. In cases where a problem would definitely occur, LabVIEW will not make a subVI out of the selection, but will present a dialog box to explain why. In cases where there is only a potential problem, a dialog box gives an explanation and lets you decide whether or not to continue.

Tip: If there are no locals or Attribute Nodes in a selection, you could use a Sequence Structure to encapsulate the code you want to a subVI and preview the results.

The following rules and recommendations will help you use this LabVIEW feature effectively.

Number of Connections

Do not make very large selections that would create a subVI with more than 28 inputs and outputs, the maximum number of connections on a connector pane.

Keep in mind that each front panel terminal, each attribute, and certain locals included in the selection require a slot in the connector pane; a selection with a large number of these items may run out of connector pane slots.

To avoid exceeding the maximum, select a smaller section of the diagram, or group data into arrays and clusters before selecting a region of the diagram to convert.

Cycles

Try to avoid making selections that could create cycles in the diagram. Cycles occur if a data flow originates from an output of the subVI and terminates as input to the subVI.

Identifying cycles while making selections is difficult, but LabVIEW will detect them for you. When cycles are detected, LabVIEW displays a dialog box asking you to either create a new VI from the selection or cancel. If you choose to create a new VI, a new Untitled VI is created from your selection. The selected items in the original diagram are left untouched.

Attribute Nodes within Loops

Do not include Attribute Nodes within a loop in your selection.

Because an Attribute Node would be retained on the caller VI and wired to the subVI, execution of the subVI would not update the value of the attribute on every iteration of the loop.

LabVIEW will not make a subVI in such cases.

Illogical Selections

Do not convert selections that do not make sense into a subVI. For example, it would not make sense to convert a selection consisting of one object inside a Sequence Structure and another object outside of the Sequence Structure without including the Sequence Structure itself.

LabVIEW disallows such selections, displaying an explanation of the problem.

Locals and Front Panel Terminals within Loops

Try to avoid including local variables or front panel terminals inside a loop in your selection.

Because selected front panel terminals or locals are retained on the caller VI and wired to the subVI, execution of the subVI will not update the value of these items on every iteration of the loop. This can cause a change in the functionality of the caller VI.

LabVIEW displays a warning in such cases and lets you decide whether to continue or cancel.

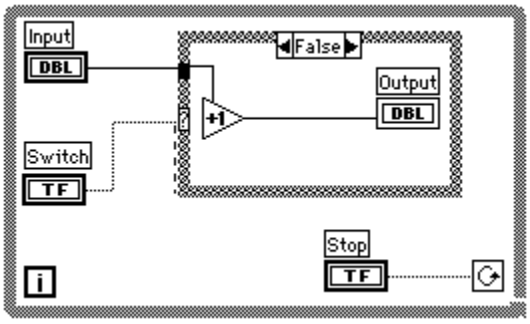
Case Structures Containing Attribute Nodes, Locals, or Front Panel Terminals

Try to avoid including a Case Structure containing an Attribute Node, front panel terminal, or local to which a value is written.

In such cases, whenever the subVI is executed, some value is always written to the object because it gets wired to the subVI, while in the original block diagram, a value can be written only when either the TRUE or FALSE part of the Case Structure is executed. This can change the functionality of the caller VI.

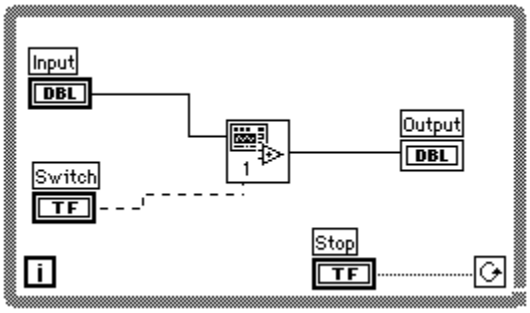
LabVIEW displays a warning in such cases and lets you decide whether to continue. If you continue, you will have to edit the subVI to supply a value in all cases.

Consider the following illustration as a simple example.



The Case Structure is selected for conversion into a subVI. The front panel terminal Output will be left

out in the original diagram and connected to the subVI through a connector as shown in the following illustration.



In the original block diagram, a value was written to Output only when the FALSE part of the case was executed. However, when the selection is turned into a subVI, a value is always written to Output regardless of whether the TRUE or FALSE part of the case was executed within the subVI. You will need to edit the subVI to supply data for the extra cases.

Hierarchy Window

The Hierarchy window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and globals.

You can configure many aspects of the display in the Hierarchy window. For example, you can have the layout displayed horizontally or vertically, and you can include or exclude VIs in `vi.lib`, globals, or type definitions.

Other useful features of the Hierarchy window include the ability to access the **VI Setup...**, **Edit Icon...**, **Get Info...**, and **Print Documentation...** options, the ability to drag or copy and paste hierarchy nodes to another VIs block diagram as a subVI, and the ability to search for hierarchy nodes by name.

Displaying the Hierarchy Window
Selecting Options for the Hierarchy Window
Finding Hierarchy Nodes

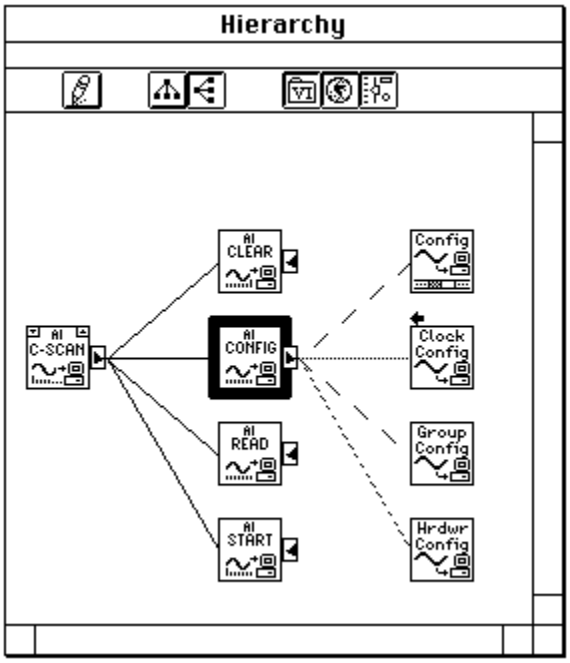
Displaying the Hierarchy Window

You can display the Hierarchy window in any of the following ways:

- Select **Project»Show VI Hierarchy**. The Hierarchy window is shown with the VI of the current active window as the focus node.
- Under the pop-up menus of subVIs, globals, or type definitions, select **Show VI Hierarchy**. The Hierarchy window is shown with the selected subVI, global, or type definition as the focus node.
- If the Hierarchy window is already open, you can bring it to the front by selecting it from the list of open windows under the **Windows** menu.

The Hierarchy window can be switched between horizontal and vertical display through an option in the **View** menu or by pressing the horizontal or vertical display button at the top of the window.

In a horizontal display, subVIs are shown to the right of their calling VIs; in a vertical display, they are shown below their calling VIs. SubVIs are always connected with lines to their calling VIs. The window shown in the following illustration is displayed horizontally.



Arrow buttons and arrows beside nodes indicate what is displayed and what is hidden, as follows:

A red arrow button pointing towards the node indicates that some or all subVIs are hidden. Clicking on the button shows the immediate SubVIs of the node.

A black arrow button pointing towards the subVIs of the node indicates that all immediate subVIs are shown.

A blue arrow pointing towards the callers of the node indicates that the node has additional callers that are not shown.

If a node has no subVIs, no red or black arrow buttons are shown.

A node becomes the focus node, indicated by being surrounded by a thick red border, whenever an operation is performed on it. It becomes de-focused when you perform an action on another node.

As you move your cursor over objects in the Hierarchy window and your cursor becomes idle over a node, the name of the node is displayed below its icon. If you prefer, you can use an option in the **View** menu to show the full path instead of the name.

You can double-click on any icon in the Hierarchy window to open the associated VI.

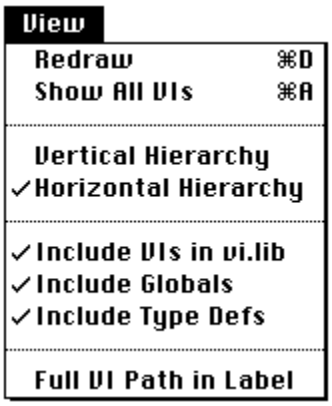
If you show the Help window and move the cursor over an icon, the Help window information for that VI is displayed.

Selecting Options for the Hierarchy Window

Several options let you control the display of the Hierarchy window, while others let you perform actions on nodes displayed in the window. Options are available in LabVIEWs **View** menu, in Hierarchy window buttons, in a nodes pop-up menu, and through mouse clicks.

Hierarchy Window View Menu Options

The **View** menu contains the following options related to the display of the Hierarchy window. Many of the options are also available from buttons near the top of the window. The **View** menu can also be accessed by popping up on the Hierarchy window.



Redraw--Redraws the window layout to minimize line crossings and maximize symmetry, which is useful after successive operations on hierarchy nodes. If a focus node exists, the window is scrolled to make that node visible. If no focus node exists, the window is scrolled to make the first root showing subVIs visible.

Show All VIs--Displays all hidden VIs. There is no focus node after this option has been selected. The option does not affect other settings in the **View** menu. In other words, VIs in `vi.lib`, globals, and type definitions that are not included in the hierarchy are still hidden.

Vertical Hierarchy--Arranges the nodes from top to bottom with the calling VIs above their subVIs.

Horizontal Hierarchy--Arranges the nodes from left to right with the calling VIs to the left of their subVIs.

Include VIs in `vi.lib`--Toggles the Hierarchy window to include or exclude VIs in `vi.lib`.

Include Globals--Toggles the Hierarchy window to include or exclude globals.

Include Type Defs--Toggles the Hierarchy window to include or exclude type definitions.

Full VI Path in Label--Toggles the Hierarchy window to display either the full VI path or just the VIs name in the tip strip for each hierarchy node.

Hierarchy Window Buttons

A toolbar in the Hierarchy window contains buttons that affect the display of the window.

The buttons, which perform some of the same actions as items in the **View** menu, are described as follows:

Redraws the window layout to minimize line crossings and maximize symmetry, which is useful after successive operations on hierarchy nodes. If a focus node exists, the window is scrolled to make that node visible. If no focus node exists, the window is scrolled to make the first root showing subVIs visible.


Arranges the nodes from top to bottom with the calling VIs above their subVIs.


Arranges the nodes from left to right with the calling VIs to the left of their subVIs.

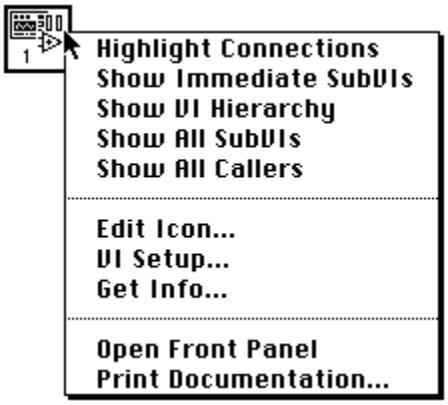

Toggles the Hierarchy window to include or exclude VIs in `vi.lib`.


Toggles the Hierarchy window to include or exclude globals.

 Toggles the Hierarchy window to include or exclude type definitions.

Hierarchy Node Pop-Up Menu

If you pop up on a node (subVI, global, or type definition) displayed in the Hierarchy window, a menu appears with options for controlling the display or carrying out commands related to the selected node.



The options are described as follows:

Highlight Connections--Makes the selected node the focus node and highlights in red the edges connecting its immediate callers and subVIs.

Show Immediate SubVIs--If the node is hiding all or some subVIs, this option expands the node to show all of its immediate subVIs. The edges connecting the node to its subVIs are highlighted in red. The option is toggled with **Hide All SubVIs**.

Hide All SubVIs--If a node is showing all immediate subVIs, this option collapses the node to hide its entire subVI chain. The option is toggled with **Show Immediate SubVIs**.

Show VI Hierarchy--Makes the selected node the focus node and displays the nodes belonging to its call chain and subVI chain. Non-related roots are also visible, but all of their subVIs are hidden. The edges connecting the nodes call chain and subVI chain are highlighted in red.

Show All SubVIs--Makes the selected node the focus node and expands its entire subVI chain. The edges connecting the nodes subVI chain are highlighted in red.

Show All Callers--Makes the selected node the focus node and expands its entire call chain. The edges connecting the nodes call chain are highlighted in red.

Edit Icon...--Displays the Icon Editor for editing of the nodes icon.

VI Setup...--Displays the VI Setup dialog box for the node.

Get Info...--Displays the Get Info dialog box for the node.

Open Front Panel--Opens the front panel of the VI, global, or type definition.

Print Documentation...--Brings up the Print Documentation dialog box, from which you can choose the portion of the VI you wish to print. For more information on this dialog box, see the Using the Print Documentation Menu Option .

Hierarchy Node Mouse-Click Actions and Shortcuts

Mouse-click sequences are available for selecting nodes, for copying or dragging nodes, and for shortcuts to certain pop-up menu options. Where applicable, the sequences are performed with the Positioning tool selected.

The shortcut mouse clicks are as follows. For a more complete description of options available in the pop-

up menu, see the Hierarchy Node Pop-up Menu topic.

Clicking on the red arrow button performs the **Show Immediate SubVIs** action.

<Shift>-clicking on the red or black arrow button performs the **Show All SubVIs** action.

Pressing <Ctrl-click> (<option-click>, <meta-click>, <Alt-click>) on the node performs the Show VI Hierarchy action.

Double-clicking on the node performs the **Open Front Panel** action.

Single-clicking on the node selects it for dragging to a block diagram or copying to the Clipboard to use as a subVI.

<Shift>-clicking and holding the mouse button down while selecting allows you to select multiple nodes for copying to other block diagrams or front panels. You can also make multiple selections by dragging a rectangle over the objects you want to select.

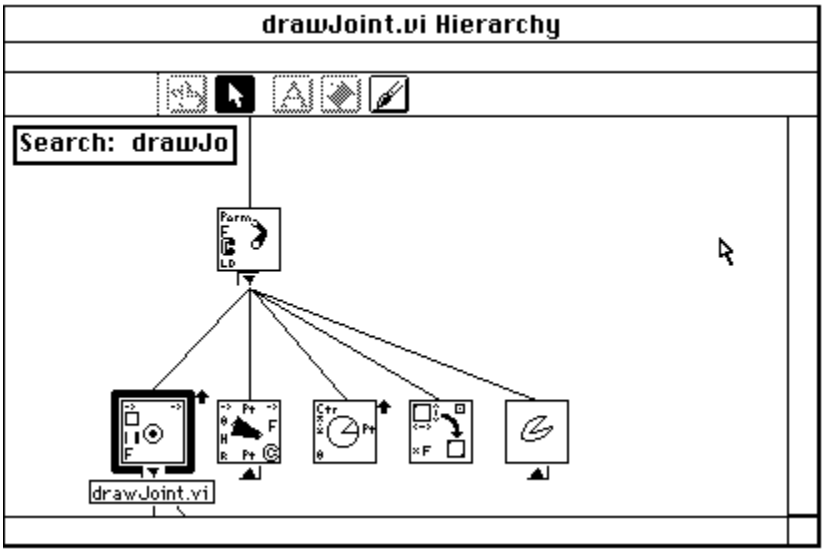
Pressing the <Tab> key toggles between the Positioning and the Scroll tool of the **Tools** palette.

Finding Hierarchy Nodes

A Find Hierarchy Node mechanism is available to search currently visible nodes in the Hierarchy window by name.

You can initiate the search by simply typing the name of the node. A small Search window appears displaying the text that has been typed. The search takes place immediately and highlights a matching node by displaying a tip strip of its name.

In the following illustration the Find Hierarchy Node mechanism has found the node named drawJoint.vi.



A search is performed as you type; thus, when no node matches the characters currently displayed in the Search window, the system beeps and no more characters can be typed. You can then use the <Backspace> or <Delete> key to delete one or more characters, so that you can resume typing.

The Search window disappears automatically if no keys are pressed for a certain amount of time. You can press the <Esc> key to remove the Search window immediately.

Once a match has been made on a node, you can use the right or down arrow key, or the <Enter> or <Return> key, to find the next node that matches the search string. To find the previous node that

matches, you can press the left or up arrow key, or the <Shift-Enter> or <Shift-Return> keys.

Using the Find Dialog Box

When developing an application consisting of multiple subVIs or even a single large VI, you may want to find occurrences of a particular object or string of text. The **Project»Find** command can help you find all instances of the following objects with the names you specify:

- VIs
- Built-in functions
- Type definitions
- Global and local variables
- Attribute Nodes
- Breakpoints
- Front panel terminals
- Text

For more information on some of the objects you can search for, see Type Definitions, Global and Local Variables, Attribute Nodes, and Breakpoints.

In addition to the Find dialog box, many objects have Find pop-up options that let you quickly find related objects. For example, if you pop up on a control, an option lets you find the corresponding terminal as well as any locals or Attribute Nodes associated with it. See Using Find Options from a Pop-Up Menu.

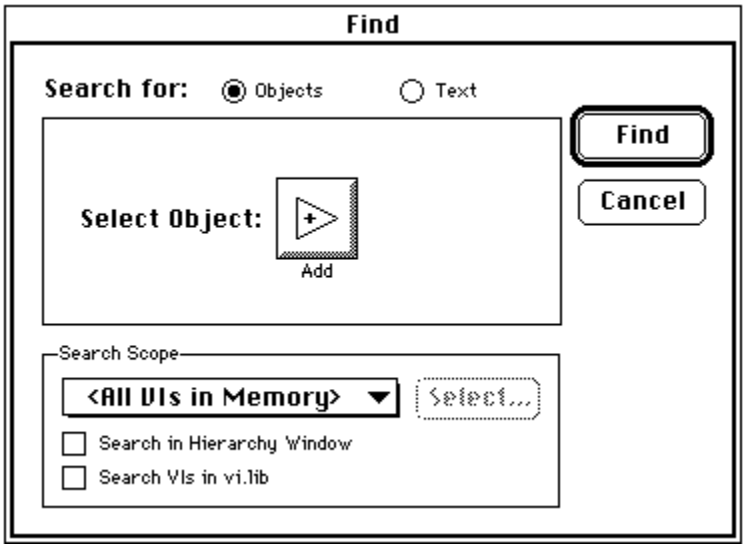
To bring up the Find dialog box, select **Project»Find...**, or press `<Ctrl-f>` (Windows); `<command-f>` (Macintosh); `<meta-f>` (Sun); or `<Alt-f>` (HP-UX).

As a shortcut, you can select a piece of text or an object before accessing the dialog box. The dialog box appears, with the text or object preselected for the search.

Finding VIs and Other Objects
Finding_Text
Selecting the Scope of the Search
Search Results Window

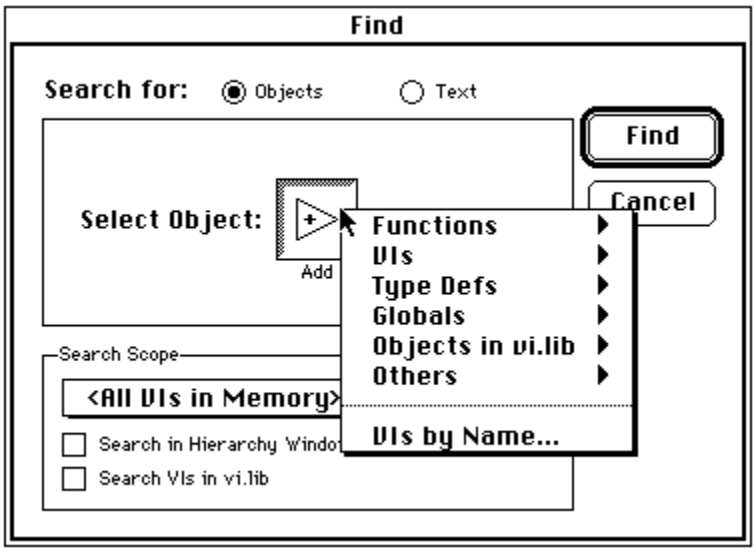
Finding VIs and Other Objects

To search for VIs or other objects, click the **Objects** button in the Find dialog box.



Specifying Objects

To select the object you want to search for, click on the button after the words Select Object. The **Select Objects** menu appears.



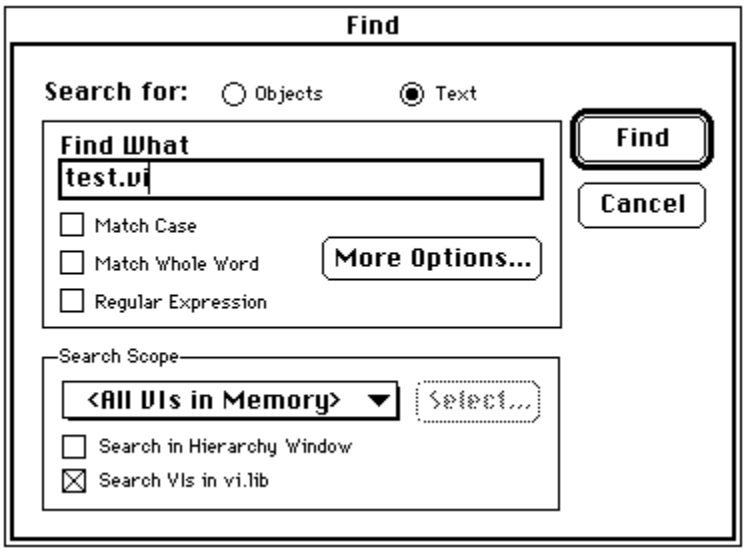
The options available in this menu are as follows:

- **Functions**--Brings up the **Functions** palette so you can select built-in functions, VIs, or any objects that you have customized to appear in the **Functions** palette.
- **VIs**--Displays a palette of all VIs in memory that are not located in `vi.lib`.
- **Type Defs**--Displays a palette of all type definitions in memory that are not located in `vi.lib`.
- **Globals**--Displays a palette of all global VIs in memory that are not located in `vi.lib`.
- **Objects in `vi.lib`**--Displays all VIs, type definitions, and globals that are located in `vi.lib`.
- **Others**--Displays a menu from which you can select Attribute Nodes, breakpoints, and front panel terminals.
- **VIs by Name...**--Invokes the Select VIs by Name dialog box, which displays all VIs in memory by name in alphabetical order. You can type the name of the object to jump quickly to the desired item in the list. The three checkboxes (VIs, Globals, and Type Defs) allow you to include and exclude different types of objects to be displayed in the list box.

For information about the scope of the search, see Selecting the Scope of the Search.

Finding Text

To search for specific text, click the **Text** button in the Find dialog box. The dialog box is displayed as follows:



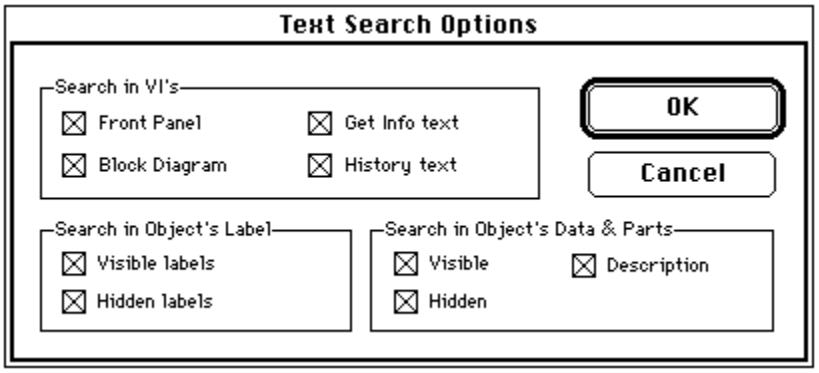
You can type in the text you want to find, and optionally limit the scope of the search to specific areas of VIs (front panel, diagram, get info text or history) or components of objects, such as the label or description.

The following options are available for specifying the search string:

- **Match Case**--Finds only text strings that exactly match the case (uppercase or lowercase) of the characters you type in.
- **Match Whole Word**--Finds text strings only if they are preceded and followed by a non-alphanumeric character such as a space or a plus sign (+), or the start or end of a line. (If this option is not selected, the command matches any string, whether it is a fragment of a larger string or not.)
- **Regular Expression**--Treats the character string as a regular expression. The regular expression has the same specifications the Match Pattern function (see the Function Reference Manual for those specifications).

Selecting More Text Search Options

The **More Options...** button invokes the Text Search Options dialog box, as shown in the following illustration.



The options are as follows:

- **Search in VIs**--Allows you to specify whether to search a VIs front panel, block diagram, Get Info text, or History window text. At least one of these must be checked.
- **Search in Objects Label**--Allows you to specify whether to search visible or hidden object name labels.

- **Search in Objects Data & Parts**--Allows you to specify whether to search visible or hidden data and parts. An objects data and parts consist of everything belonging to an object except for the objects name label. You also have the option of searching the description of various objects. At least one of the options in the Search in Objects Label and Search in Objects Data & Parts must be checked.

Note: The Text Search options are saved from search to search; therefore, if you do not find a string of text that is expected to exist, make sure that the Text Search Options are set appropriately.

For information about the scope of the search, see Selecting the Scope of the Search.

Selecting the Scope of the Search

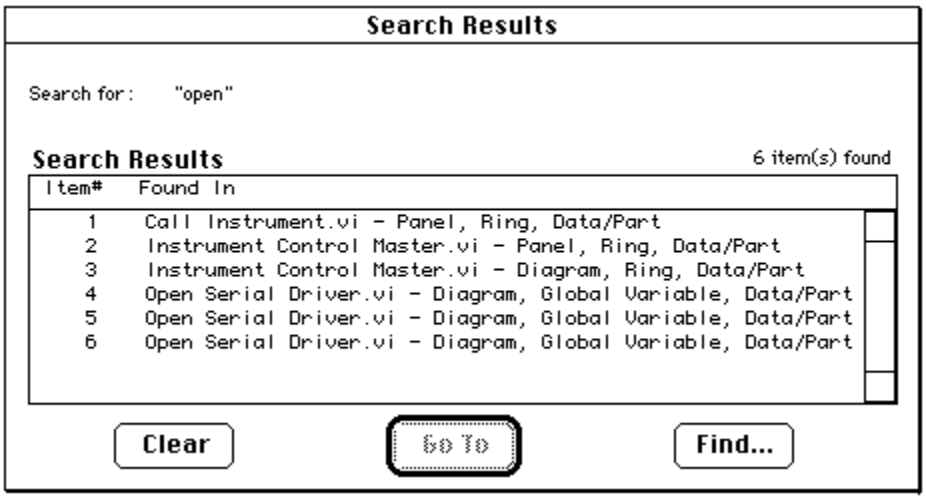
Three options in the **Search Scope** ring menu specify the VIs to be searched:

- **<All VIs in Memory>**--Specifies the search scope as all VIs that are currently loaded. Two checkboxes are available to include or exclude VIs in `vi.lib` and the Hierarchy window.
- **<Selected VIs>**--Allows you to select a custom set of VIs to search. The Hierarchy window can be included or excluded from the search scope. The Select... button brings up the Select VIs to Search dialog box.
- **Name of a VI**--Allows you to search only a specific VI. The name of the VI in this ring item depends on the active VI when you invoked the Find dialog box.

Click on the Find button to begin searching for the text or you have selected.

Search Results Window

After a search is complete, if there is more than one search result, the Search Results window, shown in the following illustration, displays all search results. If only one result is found, the result is immediately highlighted, bypassing the Search Results window.



You can display this window also by selecting **Project»Search Results....**

The following options are available in the Search Results window:

- **Clear**--Clears and frees memory used to store search results.
- **Go To**--Highlights the currently selected search result. This can also be done by double clicking the item. Note that items that have already been highlighted are checked.
- **Find...**--Invokes the Find dialog box.

- **Stop**--This button appears in place of the **Find...** button during a search. If there is a long search through many VIs, the Search Results window updates search results as they are found and shows the status of the search. The **Stop** button allows you to interrupt the search.

If an object or VI is deleted and it is part of a search result, then the item entry of that result will be grayed out. Note however that modifying text does not update the search results. Therefore you may highlight a text search result that no longer matches the search criteria. In that case, perform another find to update the search results.

Highlighting Search Results

Select **Project»Find Next** to highlight the next and previous results in the search list. For hidden text, all objects needed to display the text are made temporarily visible as grayed out objects and the text is selected; when you click the mouse or press a key, the objects become hidden again.

Using Find Options from a Pop-Up Menu

A **Find** pop-up menu option is provided on controls, front panel controls and indicators, globals, locals, and Attribute Nodes. You can pop up on a local or Attribute Node to find its corresponding control, front panel terminal, or other locals and Attribute Nodes. Conversely, controls and front panel controls and indicators provide pop-up menus to find all corresponding locals and Attribute Nodes. From a global node, you can find its corresponding global definition or all other global nodes. Conversely, global controls provide a pop-up menu for finding all corresponding global references in memory. If more than one result is found, the Search Results window appears.

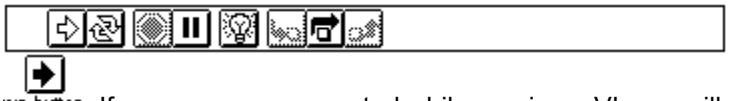
Operating VIs

Executing VIs
Data Logging in the Front Panel
Programmatic Data Retrieval
Running a VI Repeatedly

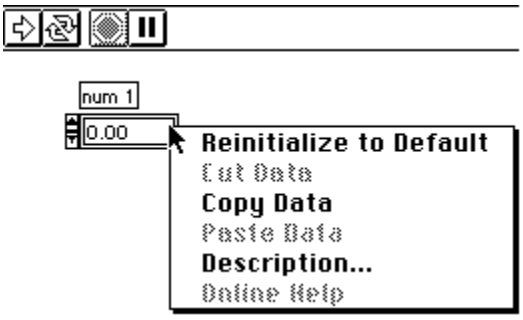
Executing VIs

When you are editing a VI, the toolbar contains several tools used for editing VIs, including the Font ring, the Alignment ring, and the Distribution ring.

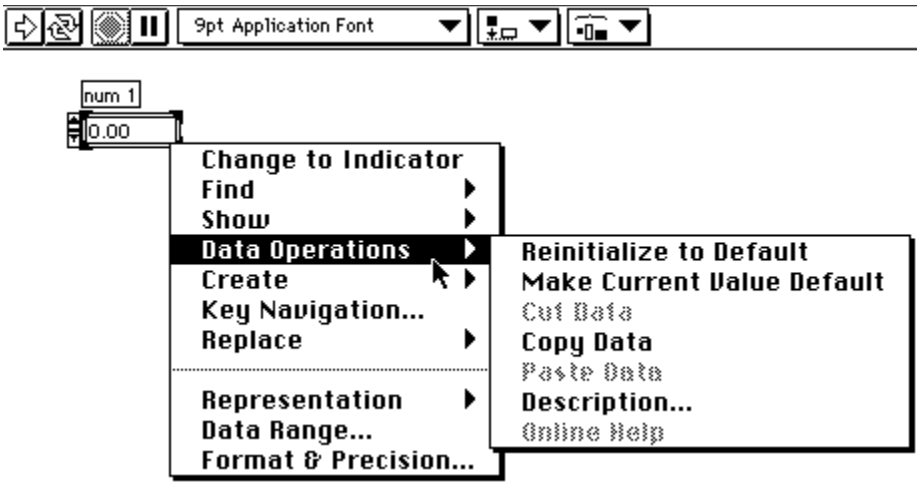
When you are running a VI, those tools are replaced with debugging tools, as shown in the following illustration.



If you pop up on a control while running a VI, you will see that all objects have a much simpler set of options. These are the same as the options in the **Data Operations** submenu that you see in pop-up menus when you edit VIs.
You can cut, copy, or paste the contents of the control, set the control to its default value, and read the description of the control with options in this menu. Some of the more complex controls have additional options. For example, an array has options you can use to copy a range of values or go to the last element of the array.



When editing a VI, an objects pop-up menu contains an editing menu and a **Data Operations** submenu, as shown in the following illustration. This submenu contains the same options you see when running a VI plus an additional option, the **Make Current Value Default** option.



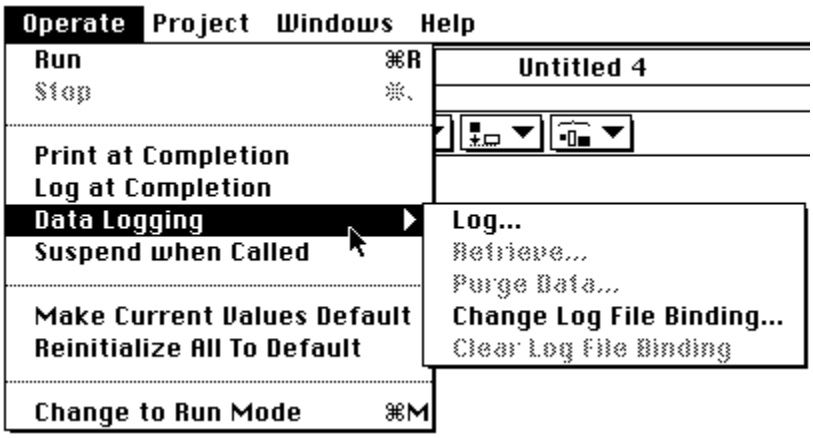
The **Operate** menu on the menu bar contains commands for executing the current VI. The following sections discuss several fundamental execution-related tasks.

Data Logging in the Front Panel

Front panel data logging logs a time stamp and the data in all front panel controls of a VI to a separate datalog file. You can have several separate files, each filled with logged data from different tests. You can retrieve this data using interactive retrieval through the VI, programmatic data retrieval, or standard file I/O functions.

Each VI maintains a log file binding, which specifies where the datalog file to which it will log front panel data resides. If the binding is clear, meaning that the location is unspecified, the VI will prompt you for the location of the file.

You configure and control front panel data logging using the **Operate** menu and its **Data Logging** submenu, as shown in the following illustration.



When automatic data logging is enabled for a VI, LabVIEW logs the VIs front panel any time the VI completes execution. You can tell whether automatic data logging is enabled for a VI by looking at the **Operate**»**Log at Completion** option. If there is a checkmark next to this option, it is enabled; otherwise it is disabled. Selecting **Operate**»**Log at Completion** enables automatic data logging for a VI if it is disabled and disables it if it is enabled.

To interactively log a VIs front panel data, select **Operate**»**Data Logging**» **Log...**.

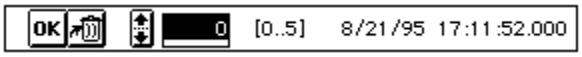
Note: A waveform chart returns only one data point at a time with front panel data logging. If you wire an array into the chart indicator, the data log will contain the array subset that was

displayed on the chart for that record.

You can change a VIs log file binding with the **Operate**»**Change Log File Binding…** option.

Selecting **Operate**»**Clear Log File Binding** disassociates the VI from any associated log file. The next time you log from the VIs front panel data, you will be prompted to specify the log file.

To view logged data interactively, select the **Operate**»**Retrieve...** option. The toolbar becomes the data retrieval toolbar, as shown in the following illustration.



The highlighted number indicates which record is currently selected for viewing. The numbers in square brackets indicate the range of records logged. The date and time indicate when the selected record was logged. You can view the next or previous record by clicking on the up arrow or down arrow, respectively. You can also use the up or down arrow on your keyboard.

You can select a specific record by typing the record number and then clicking on the enter button or pressing the <Enter> key on the numeric keypad.

You can mark the selected record for deletion by clicking on the delete button. When the selected record is marked for deletion, the delete button changes to a full trash can.

Clicking on the full trash can unmarks the selected record for deletion. Selected records are not deleted until you select **Operate»Data Logging»Purge Data...**, or until you switch out of data retrieval mode, either by clicking on the **OK** button or selecting **Retrieve...** again. If any records are still marked for deletion when you switch out of data retrieval mode, you are asked whether you want to delete the marked records.
Click the **OK** button to return to the VI whose data log you have been viewing.

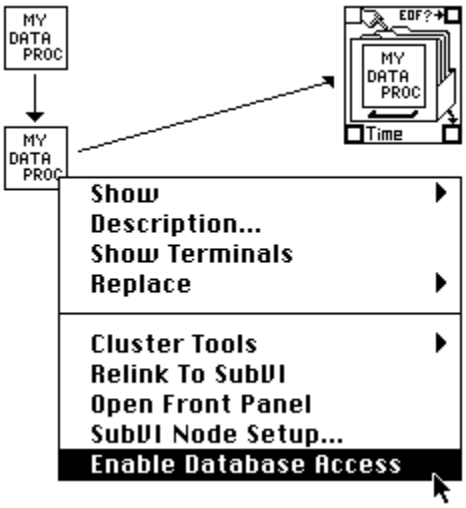
Programmatic Data Retrieval

You can retrieve data logged from a VI by using **Enable Database Access** option or using standard file I/O functions to read the file as a datalog file.

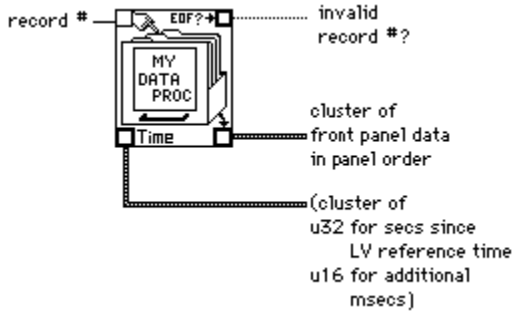
Enable Database Access
Retrieving Data Using File I/O Functions

Enable Database Access

You can also retrieve data by using the **Enable Database Access** option from the pop-up menu of a subVI that has logged front panel data in the VIs associated datalog file, as shown in the following illustration. A *halo* that looks like a file cabinet appears around the datalog file. This halo has terminals for accessing data from the datalog file. If you run the calling diagram, the subVI does not execute. Instead, LabVIEW retrieves data from a specified record of the datalog file. It also returns the time that the data was logged, and a Boolean value indicating whether the specified record number is invalid.



The following illustration shows the halo terminals for accessing the logged data.



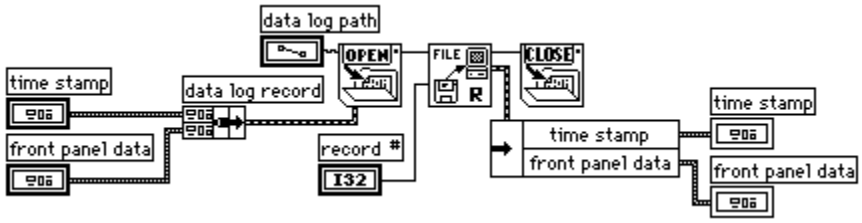
If the subVI has n logged records, you can wire any number from -n to n-1 to the record # terminal. You can access records relative to the first logged record using non-negative record numbers. 0 means first record, 1 means second record, and so on, through n-1, which stands for the last record.

You can access records relative to the *last* logged record using negative record numbers. -1 means last record, -2 means second to the last, and so on, through -n, which means the first record. If you wire a number outside the range -n to n-1 to the record # terminal, the invalid record #? output is set to TRUE, and no data is retrieved.

Retrieving Data Using File I/O Functions

You can also retrieve data that has been logged from the front panel of a VI using the LabVIEW file I/O functions. Each record in the datalog file created by front panel data logging is a cluster containing a time stamp, a cluster of an unsigned 32-bit integer for seconds since LabVIEW reference time followed by an unsigned 16-bit integer for additional milliseconds, followed by a cluster of the front panel data in panel order.

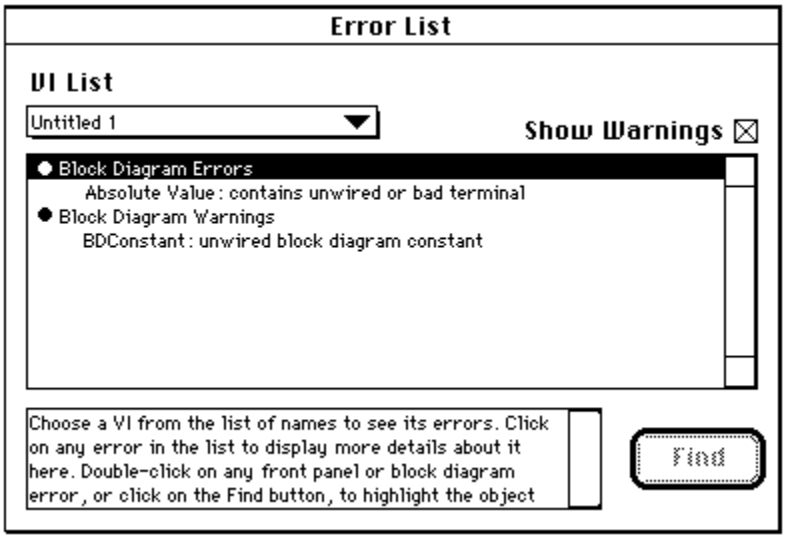
You can access the records of datalog files created by front panel data logging using the same LabVIEW file I/O functions that you use to access programmatically created datalog files. Simply provide the appropriate type, described previously, as the type input to the File Open function, as shown in the following example:


data log path
time stamp
data log record
OPEN
FILE
R
CLOSE
front panel data
record #
I32
time stamp
front panel data
time stamp
front panel data

Debugging VIs

A VI cannot compile or run if it is broken. The VI is normally broken while you are creating or editing it, until you wire all the icons in the diagram. If it is still broken when you finish, try selecting **Edit»Remove Bad Wires** or selecting <Ctrl-b> (Windows); <command-b> (Macintosh); <meta-b> (Sun); or <Alt-b> (HP-UX)**.** Often, this fixes a broken VI.


To find out why a VI is broken, click on broken run button (shown to the left). An information box called Error List appears listing all the errors. This box is shown in the following illustration.




You can also access this box by clicking on the warning button of a VI, shown at the left, or by selecting **Windows»Show Error List**. The warning button for a VI is only visible if the VI has a warning (such as overlapping objects) and you have checked the **Show Warnings** option in the Error List window. You can use the **Preferences** dialog box to configure LabVIEW to show warnings by default.
To locate a particular error, double click on the text that describes it. LabVIEW shows the error by bringing the relevant window to the front and highlighting the object causing the error. You can also view errors and warnings for other VIs by selecting the name of the VI from the **VI List** pop-up menu.

Reasons for Broken VIs
Possible Errors
Range Error Correction
Undefined Data Recognition
Debugging Techniques for Executable VIs
Single-Stepping through a VI
Highlighting Execution
Probe Tool Use
Execution Suspension
Disabling Debugging Features
Warnings
Commenting Out Sections of Diagrams

Reasons for Broken VIs

The following list contains some of the most common reasons for a VI being broken during editing.

- A function terminal requiring an input is unwired. For example, you must wire all inputs to arithmetic functions. You cannot leave unwired functions on the diagram while you run the VI to try out different designs.

- The block diagram contains a broken wire due to a mismatch of data types or a loose, unconnected end. You can remove broken wires, including wire stubs you cannot see, with the **Edit**»**Remove Bad Wires** command.
- A subVI is broken, or you edited its connector after you placed its icon in the diagram.
- You may have a problem with an object you have made invisible, disabled, or otherwise altered through an attribute node. Restore the object using the attribute node to fix the problem, if possible.

Possible Errors

The following list contains some possible errors in a form similar to the way they look in the Error List window.

- *Function Name*: `contains unwired or bad terminal`. A required input is not wired or the type is inappropriate. Wire the required inputs with the proper data types.
- `Code Interface Node: object code not loaded`. You did not link the object code of a CIN properly. Pop up on the node and reload the code.
- *Control*: `control does not match its type definition`. Edit the control to match, or pop up and update or disconnect from the type definition VI.
- `Enumeration has duplicate entries`. All the items in an enumeration control must be unique and distinct.
- `More Errors...`. LabVIEW shows only 100 errors at one time. Unlisted errors appear after you fix the listed errors.
- `For Loop: N is unwired and there are no input indexing tunnels`. Unless N is wired or an array is indexed on input, the loop cannot determine how many iterations to execute.
- `Global or Local Variable: named component doesnt exist`. The name of a variable has changed since you last loaded the VI and the name in the global or local variable no longer matches. Pop up on it and select another name.
- `Global Variable: subVI is missing`. LabVIEW could not find the global subVI when it loaded the calling VI, perhaps because you changed the name. Replace the bad global subVI with a good one.
- `Node: A subroutine priority VI cannot contain an asynchronous node`. You cannot use an asynchronous function, such as Wait, in a VI that has a priority level equal to subroutine.
- `Right Shift Register: type is undefined`. You must remove unused shift registers.
- `Right Shift Register: some but not all left sides are wired`. A shift register must have inputs for all the left sides or for none.
- `Sequence: One or more sequence locals were never assigned`. You forgot to wire to a Sequence Structure local variable. Remove unused sequence locals.
- *subVI name*: `bad linkage to subVI`. The connector pattern of the subVI changed since you last loaded the subVI, or you forgot to assign controls or indicators of the subVI to connector terminals. In the former case, you can use the **Relink** command in the subVI pop-up menu.
- *subVI name*: `recursive references (dispose it)`. You have changed the name of the calling VI to be the same as one of its subVIs, and LabVIEW now thinks the VI is calling itself recursively. (The VI and subVI are probably in different directories.) LabVIEW prevents recursion when you attempt to place a VI on its own block diagram.
- *subVI name*: `LV Subroutine link error`. A problem exists with linkage to a CIN routine.
- *subVI name*: `A Subroutine priority VI cannot call a non-subroutine priority subVI`. You must make the subVI execute at subroutine priority or change the calling VI to something

other than subroutine priority.

- *subVI name*: `subVI is already running`. You cannot run a VI if one of its subVIs is already running as a subVI of another VI.
- *subVI name:* `subVI is in either panel order or cluster order mode`. You cannot run a VI if you are changing the panel or cluster order of one of its subVIs.
- *subVI name*: `subVI is in interactive retrieval mode`. You cannot run a VI if one of its subVIs is in interactive retrieval mode.
- *subVI name*: `subVI is missing. LabVIEW could not find the subVI when it loaded the calling VI, perhaps because you changed the name`. Replace the bad subVI with a good one, or open the missing subVI if you can find it.
- *subVI name*: `subVI is not executable`. The subVI is broken. Open it and repair its errors.
- `Terminal: The associated array or cluster on the front panel has no elements; its type is undefined`. You must place a control or indicator in the array or cluster.
- `Type Definition: Cant find valid type definition`. LabVIEW could not find the type definition VI when it loaded the calling VI, perhaps because you changed the name. Open the missing type definition VI if you can find it, pop up on the bad control, and disconnect from the type definition, or replace it with a good control.
- `(Un)Bundle By Name: Empty cluster, or some components are unnamed`. The input cluster wired to the Bundle By Name function is either empty or has some components that are not labeled.
- `Unit: bad unit syntax`. The text in the node is not a legal unit expression. A `?` is placed immediately before the unrecognizable character.

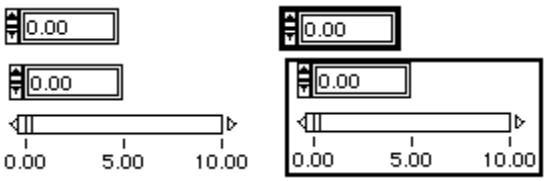
If you encounter messages that are not self-explanatory, and not listed in this section, contact National Instruments.

Range Error Correction

A range error indicator appears in place of the run button in the following circumstances.

- You configured a control on a subVI to stop execution when it receives an out-of-range value (via the **Data Range...** option of the control), and the control receives such a value.
- You configured an indicator on a subVI to stop execution when it tries to return an out-of-range value to a calling VI, and the indicator attempts to return such a value.
- An operator enters an out-of-range value into a control that you set to stop execution on error, provided that the VI is not running at the time.

You can figure out which control or indicator is out of range by its change in appearance, as shown in the following illustration.



Note: A VI or subVI can pass out-of-range values to one of its indicators without halting execution. The error condition exists only when a subVI attempts to return such a value. Also, an operator can enter out-of-range values into a control while its VI is executing. The out-of-range error condition only prevents the VI from starting, not from continuing once it has already started.

Undefined Data Recognition

There are two mnemonics that can appear in floating-point digital displays to indicate faulty computations or meaningless results. *NaN* (not a number) is the symbol that represents a particular floating-point value that operations such as taking the square root of a negative number can produce. *Inf* is another special floating-point value produced, for example, by dividing one by zero.

Undefined data can corrupt all subsequent operations. Floating-point operations propagate NaN and ± Inf, which, when explicitly or implicitly converted to integers or Booleans, become meaningless values. For example, dividing one by zero produces Inf, but converting that value to a word integer produces the value 32,767, which appears to be a normal value. Before converting to integer types, check intermediate floating-point values for validity unless you are sure that this type of error does not occur in your VI.

Executing a For Loop zero times can produce unexpected values. For example, the output of an initialized shift register is the initial value. However, if you do not initialize the shift register, the output is either the default value for the data type--0, False, or empty string--or the output is the last value loaded into the shift register when the diagram last executed.

Indexing beyond the bounds of an array produces the default value for the array element data type. You can inadvertently do this in a number of ways, such as indexing an array past the last element using a While Loop, supplying too large a value to the index input of an Index Array function, or supplying an empty array to an Index Array function.

When you design VIs that may produce undefined output values for certain input values, you should not rely on special values such as NaN or empty arrays to identify the problem. Instead, make sure your VI either produces an error message that identifies the problem or produces only defined default data.

For example, if you create a VI that uses an incoming array to auto-index a For Loop, you need to evaluate the operation of the VI if the input array is empty. You can either produce an output error code or substitute predefined values for the values created by the loop.

Note: A floating-point indicator or control with a range of -Infinity to +Infinity can still generate a range error if you send NaN to it.

Debugging Techniques for Executable VIs

If your program executes but does not produce the expected results, you can often solve the problem by taking the following steps.

- If your VI has any warnings, eliminate them. If you select the **Show Warnings** option in the Error List window, the listbox identifies the warnings for your VI. Then you can determine the causes and get rid of them.
- Check wire paths to ensure that the wires connect to the proper terminals. Triple-clicking on the wire with the Operating tool highlights the entire path. A wire that appears to emanate from one terminal may in fact emanate from another, so look closely to see where the end of the wire connects to the node.
- Use the Help window (from the **Help** menu) to make sure that functions are wired correctly.

If functions or subVIs have unwired inputs to functions or subVIs, verify that the default value is what you expect.

- Use breakpoints, execution highlighting, and single-stepping to determine whether the VI is executing as you planned. Make sure you disable these modes when you do not want them to interfere with performance.
- Use the probe feature described in the Probe Tool Use topic to observe intermediate data values. Also check the error output of functions and subVIs, especially those performing I/O.

- Observe the behavior of the VI or subVI with various input values. For floating-point numeric controls, you can enter the values NaN and ±Inf in addition to normal values.
- If the VI runs more slowly than expected, make sure execution highlighting is turned off in subVIs. Also, close subVI windows when you are not using them.
- Check the representation of your controls and indicators to see whether you are getting overflow because you have converted a floating-point number to an integer or an integer to a smaller integer. Refer also to the Undefined Data Recognition topic for more information.
- Check the data range and range error action of controls and indicators. They might not be taking the error action you want.
- Check for For Loops that may inadvertently execute zero iterations and produce empty arrays. Refer also to the Highlighting Execution and Undefined Data Recognition topics for more information.
- Verify that you initialized shift registers properly, unless you specifically intend them to save data from one execution of the loop to another.
- Check the order of cluster elements at the source and destination points. Although LabVIEW detects data type and cluster size mismatches at edit time, LabVIEW does not detect mismatches of elements of the same type. Use **Cluster Order...** option on the cluster shell pop-up menu to check cluster order.
- Check the node execution order. Nodes that are not connected by a wire can execute in any order. The spatial arrangement of these nodes does not control the order. That is, unconnected nodes do *not* execute from left to right, top to bottom on the diagram as statements do in textual languages.
- Check that there are not any extraneous VIs. Unlike functions, unwired subVIs do not always generate errors (unless you configure an input to be required or recommended). If you mistakenly place an unwired subVI on the block diagram, it executes when the diagram does, and consequently your program may end up doing extra actions.
- Check that you do not have hidden VIs. You may have inadvertently hidden subVIs three ways--by dropping one directly on top of another node; by decreasing the size of a structure without keeping the subVI in view; or by placing one off the main diagram area. In the last case, scroll the block diagram window to its limits. Also check the inventory of subVIs used in the VI against the **Project** menu options **This VIs SubVIs** and **Unopened SubVIs** to determine whether any extraneous subVIs exist. You can also look in the Hierarchy window to see a VIs subVIs, or in the Error List window to see a list of hidden objects (as long as the **Show Warnings** checkbox is selected). To help avoid corruption caused by hidden VIs, you can specify that inputs to VIs are required; for information, see the Required Connections for SubVIs topic.

Single-Stepping through a VI


For debugging purposes, you may want to execute a block diagram node by node, called single-stepping. While executing a VI, you can start single-stepping by clicking on the pause button.


If you have not yet started executing a VI, you can start single-stepping by pressing the start single-stepping button. From single-stepping you can return to normal execution at any time by clicking on the pause button (which has actually become the continue button).
You may want to use execution highlighting as you single-step through a VI, so that you can follow data as it flows through the nodes. See the Highlighting Execution topic for more information.

While the VI is running in single-step mode, you can press any of the three step buttons that are active to proceed to the next step. The step button you press determines where the next step will be executed.

If you idle your cursor over any of the step buttons, a tip strip will appear with a description of what the next step will be if you press that button.

Step Buttons

Keyboard Commands
Viewing the Call Chain
Single-Stepping through a VI Example

Step Buttons

The step buttons are as follows:

 Press the step over button to execute a structure (sequence, loop, etc.) or a subVI and then pause at the next node. The keyboard shortcut is <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) followed by the right arrow key.

 Press the step into button to execute the first step of a subVI or structure (sequence, loop, etc.) and then pause at the next step of the subVI or structure. The keyboard shortcut is <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) followed by the down arrow key.

 Press the step out button to finish executing the current block diagram, structure, or VI and pause. The keyboard shortcut is <Ctrl> (Windows); <command> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) followed by the up arrow key.
When the VI finishes executing, the step buttons become gray.

The step buttons affect execution only in a VI or subVI that is in single-step mode. If a VI in single-step mode has one subVI that is also in single-step mode and one that is in normal execution mode, the first subVI single-steps when called but the second executes normally when called.

Keyboard Commands

In addition to the keyboard shortcuts given in the Step Buttons topic, the following commands are available:

- When single-stepping through a subVI or structure, clicking on the step out button while holding the mouse down brings up a menu from which you can select how far the VI should execute before pausing.
- Double-clicking on front panel controls and indicators, local variables, and global variables displays and highlights the corresponding block diagram object.
- Pressing the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key while double-clicking on a subVI brings its block diagram to the front.

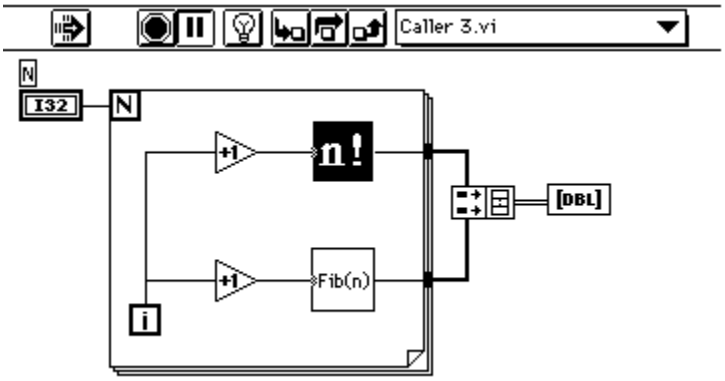
Viewing the Call Chain

When a subVI is paused, a Call Chain ring appears. This menu lists the chain of callers from the top level VI down to this subVI. Note that this is not the same as the **Project»This VIs Callers** option, which lists all calling VIs regardless of whether they are currently executing. When you select a VI from the Call Chain ring, its block diagram opens and the VI calling the current subVI is selected. This helps you distinguish the current instance of the subVI if the block diagram contains more than one instance.

Single-Stepping through a VI Example

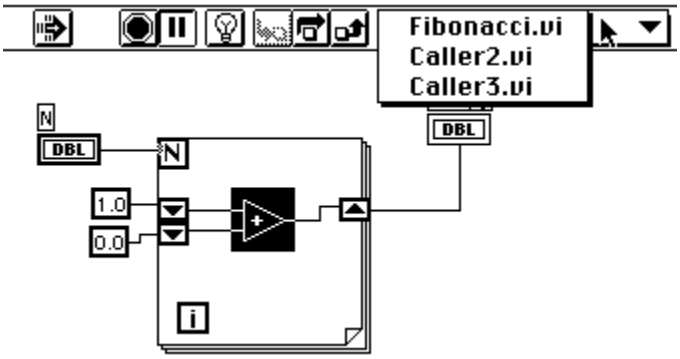
The following illustration shows an example of a VI that is single-stepping (Caller 1.vi). This VI is currently running on behalf of the VI Caller 3.vi. The subVI Fibonacci.vi is currently executing (as can be seen by the arrow glyph on the run button). The VI Factorial.vi is the next node to be executed. Clicking on the step into button will open its block diagram and start single-stepping. Clicking on the step

over button will execute the subVI and pause. Clicking on the step out button will finish executing the current frame of the loop and pause.



After stepping into the subVI `Fibonacci.vi` and stepping into the loop, clicking on the step out button and holding down the mouse for a second, a menu appears from which the user can select how far the VI should execute before pausing. Selecting **Block Diagram** will run the VI until all nodes on the VIs block diagram have executed, at which time execution will pause.

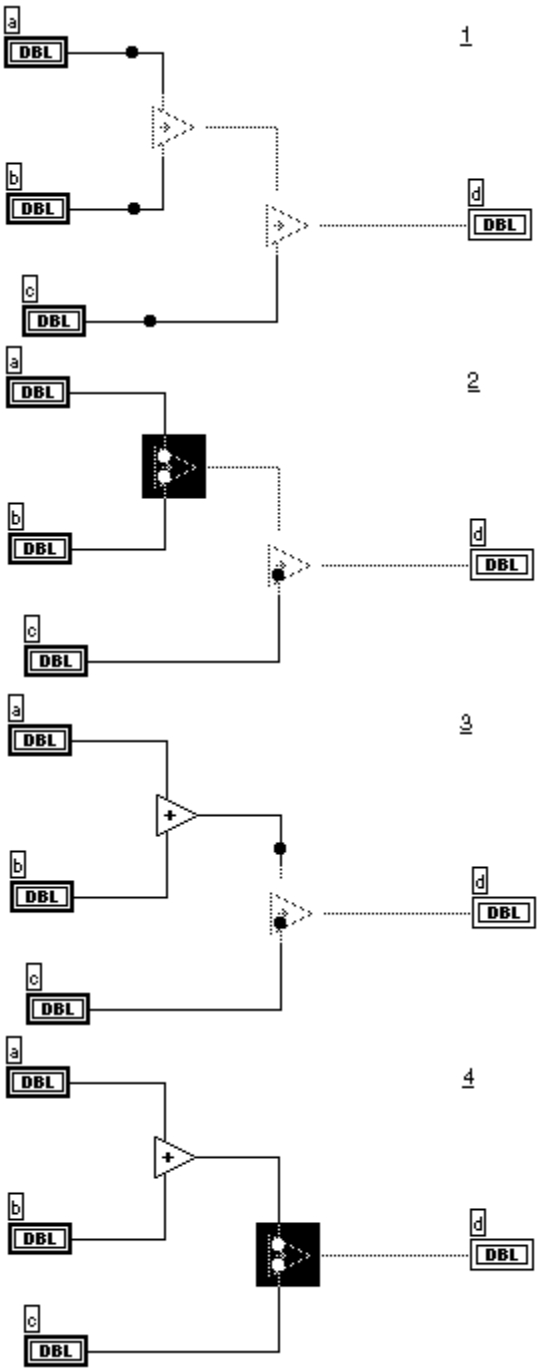
Selecting one of the VIs on the call stack will run that VI until it is finished. At that point, the selected VIs caller will be paused.



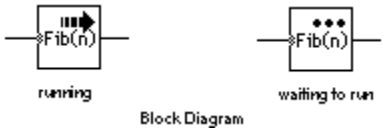
Highlighting Execution

For debugging purposes, it is helpful to view an animation of the execution of the VI block diagram.

To enable this feature, click on the hilite execution button. The button changes its appearance. Click on this button at any time to return to normal view mode. You commonly use hilite execution in conjunction with single-step mode to gain an understanding of how data flows through nodes. Highlighting greatly reduces the performance of a VI.
With execution highlighting, the movement of data from one node to another is marked by bubbles moving along the wires. In addition, in single-stepping, the next node blinks rapidly as shown in the following illustration sequence.



When you are single-stepping through a subVI with hilite execution on, an execution glyph on the block diagrams subVI icon indicates which VIs are running and which are waiting to run.



The Hierarchy window displays whether a VI is paused and/or suspended. The arrow in the following illustration indicates a subVI that is running regularly or single-stepping. The pause glyph indicates a subVI that paused and/or suspended. A green pause glyph (or a hollow glyph if displayed in black and white) indicates that this subVI will pause when called. A red pause glyph (or a solid glyph if displayed in

black and white) indicates that this subVI is currently paused.

Hierarchy Window

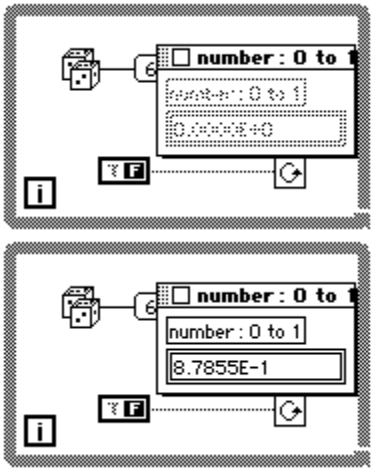
Probe Tool Use

You can use the Probe tool, selected in the illustration below, to check intermediate values in a VI that executes but produces questionable or unexpected results. For example, you may have a complicated block diagram with a series of operations, any one of which may be returning incorrect data.

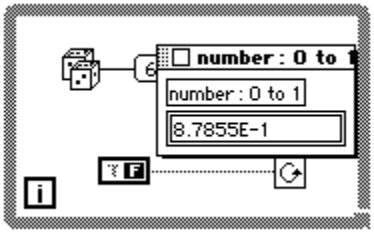
Probe tool selected

One way you could look for the source of the questionable results would be to wire an indicator to the output wire from one of the operations to display the intermediate results. But placing an indicator on the front panel and wiring its terminal to the block diagram is not a convenient debugging mechanism. It is time-consuming and creates unwanted items on your front panel and block diagram that you must later delete.

Select the Probe tool and place its cursor on a wire, or pop up on the wire and select Probe. As shown in the following illustration, the floating Probe window appears, with no data displayed within.



As soon as you run the VI, the Probe window displays data passed along the wire, as shown in the following illustration.



Each probe is automatically and uniquely numbered and the wire to which it is attached is marked with the same number. This helps you keep track of which probe is associated with which wire. However, if the

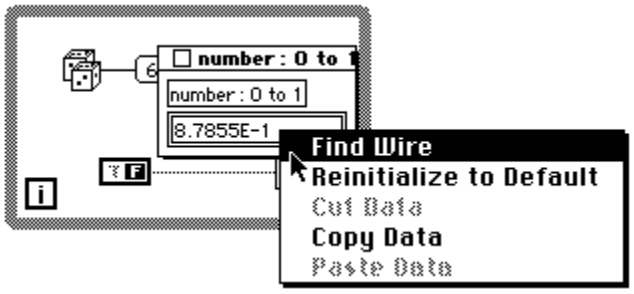
name of the front panel control is as long as or longer than the Probe window, the number will not be visible.

You can use the probe in conjunction with execution highlighting, single-stepping, or breakpoints to view values more easily. During single-stepping or pausing at a breakpoint, data is immediately updated if it is available. This allows you to examine a nodes inputs when execution halts at the node.

You can insert the probe before running your VI in order to see the data. In single-stepping, you can also create a probe for a wire that has just executed and it will update to show the wires contents. This is useful when you have set a breakpoint on a node and you want to examine its inputs.

You cannot change data with the probe, and the probe has no effect on VI execution.

If you pop up on the indicator of the Probe window, you can find the associated wire, by selecting Find Wire, as shown in the following illustration.



If you select the **Find Wire** option, the block diagram containing that wire comes to the front of all VIs and the wire is highlighted.

Use **Reinitialize to Default** to reset the value displayed in the Probe window to its default value.

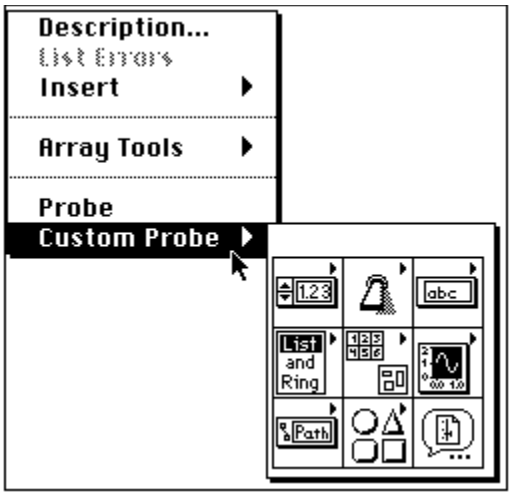
Use the **Copy Data** option to copy the data to other numeric controls in the same VI or in other VIs.

Click here to access the Probe Options topic.

Probe Options

When you create a probe, LabVIEW chooses creates a default-style probe to match the data type of the wire. With numeric data types, for example, LabVIEW uses a digital indicator to probe the data of a wire.

If you prefer, you can select a control for the probe from the built-in controls, or from controls that have been saved as custom controls or type definitions. To do so, pop up on the wire, select **Custom Probe**, and then select a control from the **Custom Probe** palette that appears to the right. If you choose **Select a Control…**, you can use the file dialog box to select any custom control or type definition that was saved in the file system. Type definitions are treated just like standard custom controls when used to probe data.



Only those parts of the **Custom Probe** palette that might match the data type of the wire are enabled. If you choose a control that cannot have the same data type as the wire, LabVIEW beeps. For example, with arrays and clusters, you cannot use the array and cluster shells from the **Array & Cluster** palette, because they are not completed data types.

You can add your own palettes to the end of the **Custom Probe** palette, just as you can add them to the **Controls** palette. Customizing the Controls and Functions Palettes for more information on how to do this.

Setting Breakpoints

You can place a breakpoint on a VI, node, or wire to set a pause to occur during execution. Nodes include subVIs, functions, structures, Code Interface Nodes (CINs), Formula Nodes, and Attribute Nodes. When a block diagram has a breakpoint, execution pauses after all nodes on the diagram have executed. When a wire has a breakpoint, execution pauses after data has passed through the wire. The following illustration displays the breakpoint options.

When you reach a breakpoint during execution, you can single-step through execution, probe wires to see their data, change values of front panel controls, or continue running to the next breakpoint or until execution is completed, whichever comes first.

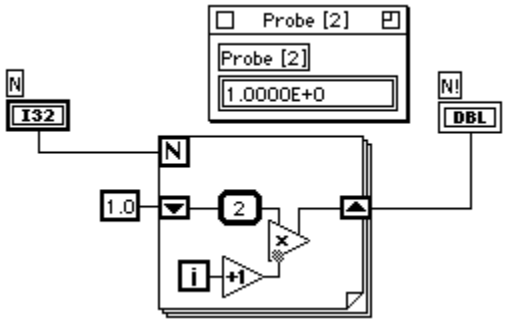
To set a breakpoint, select the Breakpoint tool in the **Tools** palette and then click the block diagram, node, or wire. Click this tool on the same object at any time to remove the breakpoint. The appearance of the tool indicates whether a breakpoint will be set or cleared, as shown at the left.

The following table indicates how breakpoints are displayed and when execution will pause, in relation to where breakpoints are placed.

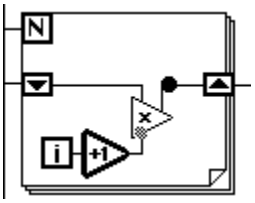
Location of Breakpoint	How Breakpoint is Highlighted	When Pause Will Occur
Block	Red border around	When the end of the

diagram	block diagram. If the diagram is inside a structure, the red border is inside the structure as well.	block diagram is reached.
Node	Red border framing the node.	Just before the node executes. At this point all the input signals into the node can be probed with the Probe tool.
Wire	Red bullet in the middle of wire. If a probe is attached to the wire, the probe has a red border.	After data has passed through the wire.

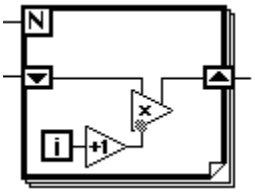
In the following example, the wire with the attached probe has a breakpoint. The VI will pause after the probe has generated data.



In the next example, the Increment node has a breakpoint. The VI will pause before Increment executes. One wire also has a breakpoint. The VI will pause after Multiply has executed.



And in the following example, the For Loop has a breakpoint. The VI will pause after Multiply has executed.



When a VI pauses because of a breakpoint, its block diagram is brought to the front (if the front panel was closed, it is opened as well), and the node or wire that caused the break is highlighted with a marquee.

Breakpoints are saved with a VI, but only become active during execution.

Execution Suspension

Suspending execution of a subVI lets you edit indicators, execute the subVI as many times as you want before returning to the caller, or go back to the beginning of the subVI.

To set a subVI in suspend mode, open the subVI and check the **Operate**»**Suspend when Called** option. This option can also be accessed by popping up on the connector pane on the front panel while in run mode, and selecting **Setup...**»**Execution Options**»**Suspend When Called...**. The subVI will then be automatically suspended whenever it is called. If you check this option when single-stepping, the VI will switch into suspend mode immediately.

If you want to suspend execution at a particular call to a subVI, use **SubVI Node Setup...** option from the subVI nodes pop-up menu, instead of the **Suspend when Called** option. The **SubVI Node Setup...** suspends execution at that particular instance of the subVI only.

Automatic SuspensionSuspended SubVI Button Use
Hierarchy Window During Suspension

Automatic Suspension

A VI is automatically suspended if, during its execution, a control or an indicator goes out of range. The terminal or local variable that caused the out of range is selected.

A subVI with a control or indicator set to stop on a range error has a conditional breakpoint. If a range error occurs, the subVI pauses as if it encountered a breakpoint. If no range error occurs, the subVI executes normally.

When data passed to a subVI causes a range error, the subVIs front panel opens or comes to the foreground and the subVI remains in suspend mode. At this time, the values on the subVI controls are the inputs passed by the calling VI, and you can change the values if you want. In fact, you must change them to run the subVI if the range error indicator is on. The indicators of the subVI display either default values or values from the last execution of the subVI during which its panel was open.

Suspended SubVI Button Use

If you want to execute the current subVI before returning to the caller, press the run button (or select the **Operate**»**Run** command) while in suspend mode. You can repeat the execution as many times as you want.
When the subVI completes, the indicators display results from that execution of the subVI. However, you can change the indicator values if you want to return different values to the calling VI. In fact, the suspend mode is the only time you can set values on indicators. Click on the run button again when you are ready to return the indicator values of the subVI to the calling VI.

If you want to go back to the beginning of the VI, click on the skip to beginning button.

The return to caller button appears when a suspended subVI is idle. Click on it to return to the caller VI. Notice that you can return to a caller without executing the current VI. If you want to execute it, be sure to press the run button before returning to the caller.

Hierarchy Window During Suspension

The Hierarchy window displays an exclamation point glyph to indicate a subVI that has been suspended. The following illustration shows a subVI in the Hierarchy window with the **Suspend when Called** option is

on.

Disabling Debugging Features

To reduce memory requirements and slightly increase performance, you can compile a VI without various debugging features. To do this, go to **VI Setup...»Window Options** from the connector pane pop-up menu on the front panel and deselect the checkboxes for **Show Run Button**, **Show Continuous Run Button**, **Show Abort Button**, or **Allow Debugging**.

Commenting Out Sections of Diagrams

In some cases, you may want to execute a VI with a section of the block diagram disabled. The easiest way to do this is to enclose the section of diagram that you want disabled in a Case Structure with a constant Boolean wired to the selector. Put the diagram you want to disable in the frame of the Case Structure that does not execute.

If the Case Structure produces any values, you need to provide a value for the output tunnels.

Warnings

If an object is completely hidden by another object, LabVIEW generates a warning. For example, if a terminal is hidden under the edge of a structure, or tunnels are on top of each other, a warning message is placed in the Error List window. Warnings do not prevent you from running a VI; they are just intended to help you to debug potential problems in your programs.

Running a VI Repeatedly

 To execute a VI repeatedly, click on the continuous run button. The VI begins executing immediately, and the continuous run button changes from outlined arrows to filled arrows while the VI is running. Click on the continuous run button again to stop the VI. The VI stops when it has completed normally.

 The behavior of the VI and the state of the toolbar during continuous run is the same as during a single run started with the run button or the **Operate»Run** command.

Creating Pop-Up Panels and Setting Window Features

This topic discusses how you can use the VI Setup and SubVI Node Setup dialog boxes to customize the behavior of a subVI. You can use these dialog boxes to create subVIs that show their panels when they are called and to enable and disable features of a panel, such as the scrollbars, the toolbar, and the window resizing capability.

The topic contains the following subtopics:

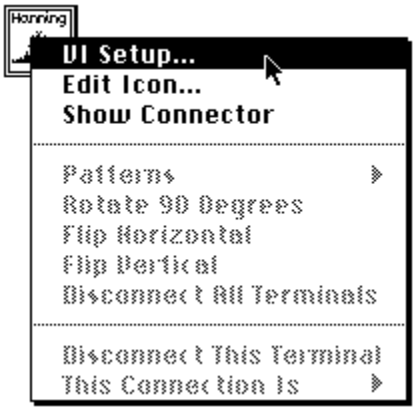
Creating a Pop-Up Panel
Setting Execution Options
Setting Documentation Options
Setting Window Features
SubVI Setup Dialog Box

Creating a Pop-Up Panel

A single front panel is sometimes too restrictive if you need to present numerous options or displays. The best solution to this problem is to organize your VIs so that high level options are presented in your topmost VI, and related options are presented by subVIs.

When a subVI is called, it ordinarily runs without opening its front panel. You can use the VI Setup or SubVI Node Setup dialog boxes to make a subVI open its front panel when the subVI is called and close the panel when the subVI is finished running.

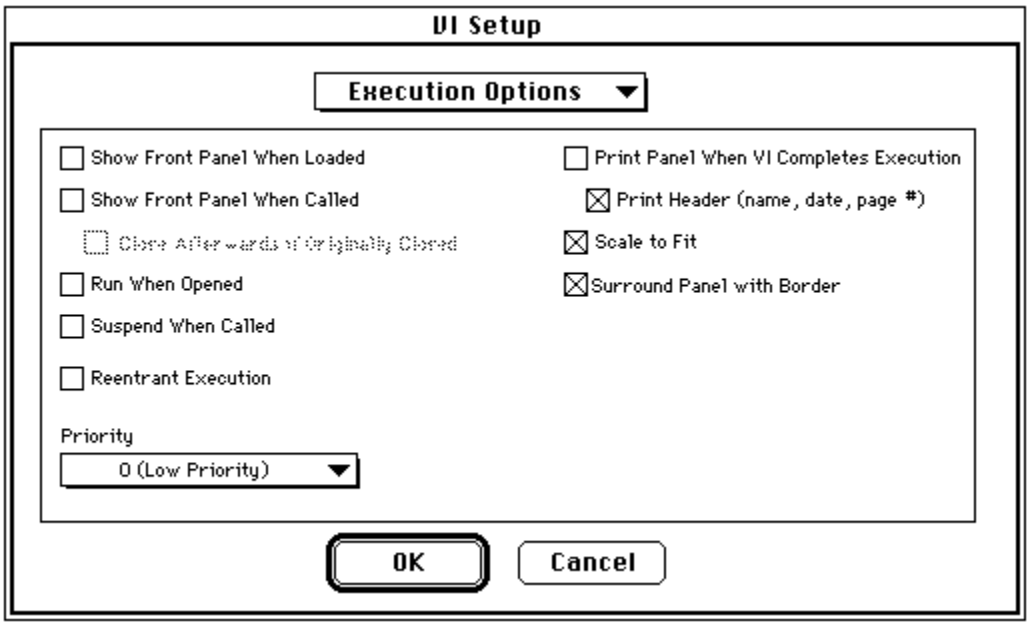
You access the VI Setup dialog box by popping up on the VI icon in the top-right of a front panel and selecting **VI Setup…**, as shown in the following illustration. You must be in edit mode.



This dialog box contains several options for customizing the look and behavior of the VI. Use the ring at the top of the dialog box to select from three different option categories--**Execution Options**, **Window Options**, and **Documentation**.

Setting Execution Options

Initially, the VI Setup dialog box presents you with execution options for the current VI. The **Execution Options** are shown in the following illustration.



If you want a subVI panel to open immediately, as soon as the top level VI is loaded, turn on the **Show Front Panel When Loaded** option.

Two of the most useful options of this page are **Show Front Panel When Called** and **Close Afterwards if Originally Closed**. If you turn these options on for a subVI, that subVI panel opens automatically when the subVI is called. See the *LabVIEW Tutorial* for some examples of how this works.

The **Run When Opened** option is another useful option you can use to set up a VI to start running automatically when it is opened.

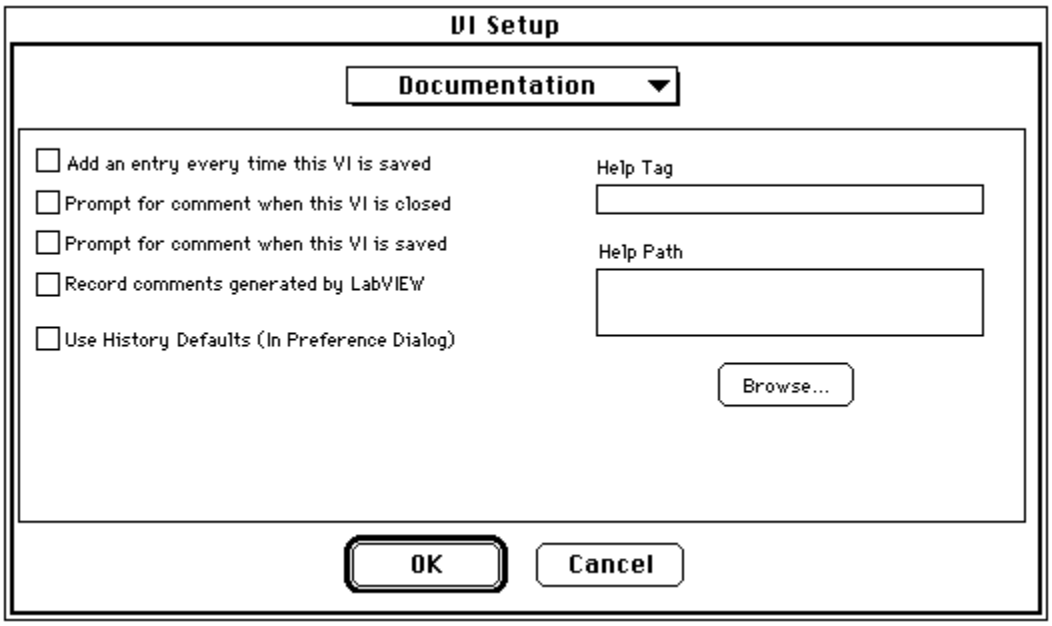
Setting the **Suspend When Called** option is the same as selecting **Operate»Suspend when Called**. This debugging option is described in the LabVIEW DebuggingVIs topic.

The **Reentrant Execution** and **Priority** options are fairly advanced features that affect the way a VI executes. You only need them in special applications. These options are discussed in the Reentrant Execution and VI Setup Priority Setting topics.

You can use the remaining options to set your VI to print when executed and customize the way it looks when printed in this way. Programmatic printing is discussed in detail in Programmatic Printing.

Setting Documentation Options

The dialog box in **VI Setup…»Documentation**, shown in the following illustration, gives you options concerning entries made to the History window, which displays the development history of the VI. For information on the History window, see the History Window topic.



To use any of the first four options in the dialog box of **VI Setup…»Documentation**, the **Use History Defaults (In Preferences Dialog)** option must be deselected. If the option were selected, LabVIEW would use the history preferences instead. The history preferences are identical, except that they set up the defaults used when you create a new VI, whereas the VI Setup documentation options apply only to the current VI.

If you check the **Add an entry every time this VI is saved** option, LabVIEW will add to the VI history every time you save the VI. If you have not entered a comment in the **Comment** box of the History window, only a header will be added to the history. (The header will contain the revision number if that option is checked in the **Preferences»History** dialog box, the date and time, and the name of the VI.)

If you check the **Prompt for comment when this VI is closed** option, the History window will appear so that you can enter a comment whenever you close a VI that has changed since you loaded it, even if you have already saved the changes.

If you check the **Prompt for comment when this VI is saved** option, the History window will appear whenever you save so that you can enter a comment. This is useful if you prefer to comment on your changes when you finish making them instead of as you are editing. If you do not set this option, you will not have a chance to change the history of the VI after you select **Save** until the save is finished.

Note: You will not be prompted to enter a comment when you save or close a VI if the only changes you made were changes to the history.

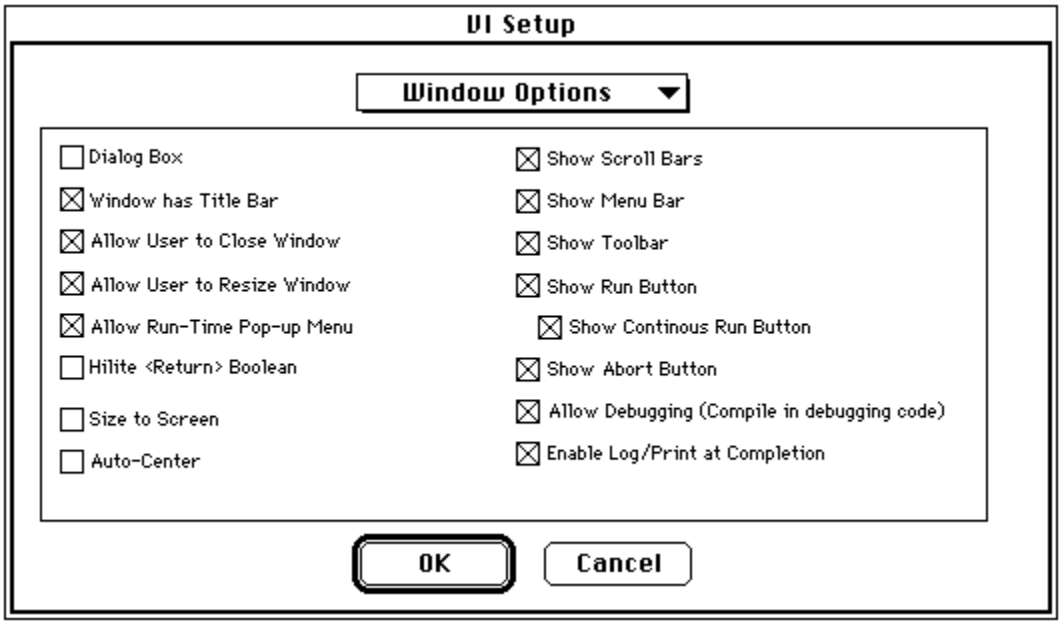
The **Record comments generated by LabVIEW** option causes LabVIEW to insert comments into the History window when certain events occur. The events that cause automatic comments are conversion to a new version of LabVIEW, SubVI changes, and changes to the name or path of the VI.

The options to the right of the **VI Setup…»Documentation** dialog box give access to online help files that you have created. Put your cursor in the **Help Tag** box and type a topic to display for this VI. Then type the path to the help file in the **Help Path** box, or click the **Browse...** button and find the file. The filename and path of the file appears in the **Help Path** box. After you have set these options, clicking on the online help icon at the bottom of the Help dialog box will access the help file you selected. the Creating Your Own Help Files topic.

Setting Window Features

The window options apply to the VI when it is in run mode, but not in edit mode. You can use these options to control a user's ability to interact with the program by restricting access to LabVIEW features, and by changing the way the window looks and behaves. You can easily make your VI look and act like a dialog box, in that the user cannot interact with other windows while this window is open. You can also remove the scrollbars and the toolbar, and set a window to automatically be centered or sized to fit the screen.

The window options are shown in the following illustration.



Dialog Box prevents the user from interacting with other LabVIEW windows while the window with this option selected is open, just as a system dialog box does.

If you turn off the **Allow User to Close Window** checkbox in the VI Setup dialog box, the Window Options dialog box hides the close box for the VI front panel, and dims the **File»Close** option. With this VI setup option, you can ensure that an operator cannot inadvertently close a run-time or pop-up VI front panel.

Allow Run-Time Pop-Up Menu determines whether objects on this front panel can display a pop-up menu of data operations in run mode.

When you select the **Hilite <Return> Boolean** option, LabVIEW highlights any Boolean currently associated with the `<Enter>` (Windows and HP-UX), or `<Return>` (Macintosh and Sun) key. You associate keys with controls using the Key Navigation dialog box, which is described in the Key Navigation Dialog Box topic.

The **Size to Screen** option automatically resizes the panel of a VI to fit the screen when the VI is switched to run mode and when it is loaded into memory. Notice that the VI does not remember its original size and location, so it stays in the same place if you switch back to edit mode.

Auto-Center automatically centers the front panel on the user's computer screen when the VI is opened or when it is switched to run mode.

Other options toggle between showing and hiding various window features. You can hide individual buttons by deselecting them, or hide the entire toolbar by deselecting the **Show Toolbar** option.

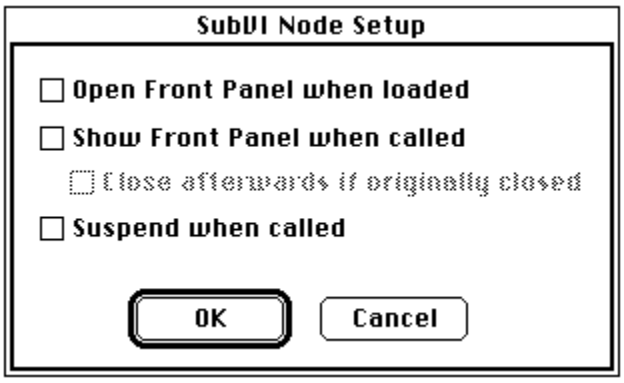
Enable Log/Print at Completion enables or disables automatic data logging (see Data_Logging_in_the_Front_Panel and Programmatic Printing.

Caution: If you hide the menu bar and the toolbar, there is no visible means of changing from run mode back into edit mode. Use the <Ctrl-m> **(Windows);** <command-m> **(Macintosh);** <meta-m> **(Sun); or** <Alt-m> **(HP-UX) hot key that corresponds to the Operate»Change to Edit Mode option to change the VI back to edit mode, where the menu bar and palette are visible.**

The options do not take effect until the VI starts running.

SubVI Setup Dialog Box

The following dialog box appears when you select **SubVI Node Setup...** from the node pop-up menu of a subVI on the block diagram of another VI.



These options are a subset of the options in the VI Setup dialog box. The difference between the two sets of options is that you use the SubVI Setup dialog box to specify options that relate to a specific call to a subVI, whereas you use the subVI execution options of VI Setup to specify the behavior of all calls to that VI.

Printing in LabVIEW

There are three primary kinds of printing in LabVIEW.

- You can use the **Print Window** option to make a quick printout of the contents of the current window. For information, see PostScript Printing.
- You can make a more comprehensive printout of a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on, by selecting the **Print Documentation** option. For information, see Using the Print Documentation Menu Option.
- You can use the LabVIEW programmatic printing feature to make VIs which can print under the control of your application. For information, see Programmatic Printing.

Some additional techniques address various printing concerns. For information, see Other Methods for Printing.

See also Setting Up LabVIEW to Print.

Setting Up LabVIEW to Print

There are two dialog boxes that affect the appearance of all printouts, regardless of the method you use to print.

You use LabVIEW's **Preferences»Printing** dialog box to tell LabVIEW how it should print. This dialog box, for example, gives you the option of using PostScript printing. For information, see Printing Preferences.

To configure information and formatting that is printer specific, use the **Printer Setup** (**Page Setup** on the Macintosh) option from the **File** menu. For example, with most printers, you change the orientation of printouts (landscape versus portrait) using this dialog box. Many printers also have options for font substitution, paper size, and other printer specific settings. **Printer Setup** (**Page Setup** for the Macintosh and UNIX) settings are saved with your VI.

PostScript Printing

To get higher quality printouts, you can use the Preferences dialog box, which contains options for customizing the way LabVIEW prints, including the option to select PostScript printing (if you have a PostScript printer).

If you have a PostScript printer, you can take advantage of the following benefits.

- PostScript printouts reproduce the image of the screen more accurately.
- PostScript facilitates high resolution graphs.
- PostScript reproduces patterns and line styles more accurately.

You use the Preferences dialog box to specify whether or not you want to use PostScript printing. If you select PostScript printing, you have the option to select Level 1 PostScript, or Level 2 PostScript, which supports color printing. For information, see Printing Preferences for information on the **Preferences»Printing** dialog box.

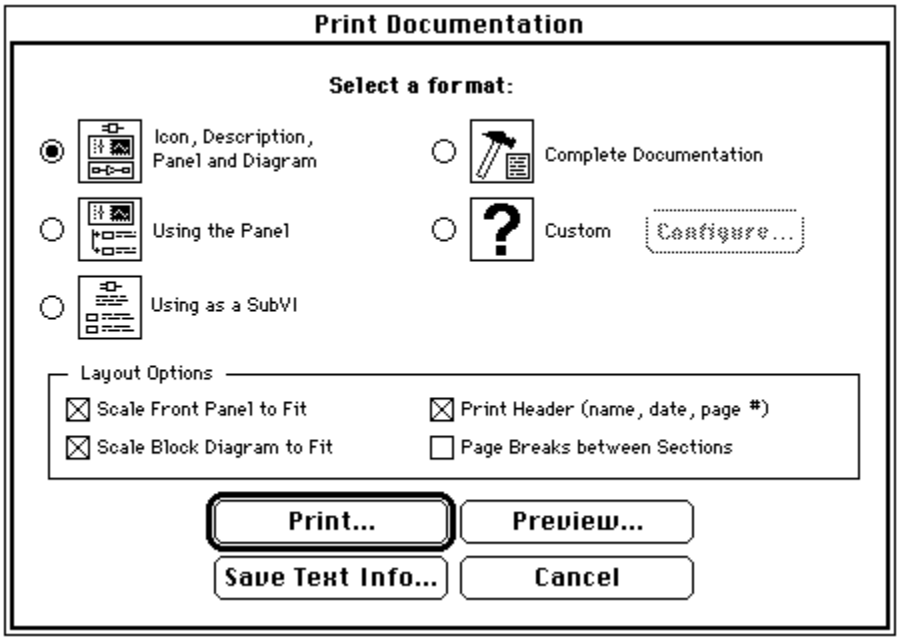
Using the Print Window Menu Option

You can use the **Print Window** option to print out the contents of the currently active window (front panel or block diagram). Using this option, you can make a quick printout with the minimum number of prompts. For more comprehensive printouts of VIs, you should use the **Print Documentation** menu option.

Print Window formats the printout so that it looks the same as it would if you set the VI to print programmatically. The **Execution Options** section of the VI Setup dialog box contains a few options which give you more control over the way your VI looks when you print programmatically or when you print using **Print Window**. See the Setting up the Page Layout for information on these options.

Using the Print Documentation Menu Option

You should use the **Print Documentation…** option, shown in the following illustration, if you want a more detailed printout of the contents of a VI. This option displays a dialog box you can use to select from several VI formats. You can also create your own custom format.



Printout Formats
Print Documentation Layout Options
Custom Print Settings

Printout Formats

The Print Documentation dialog box has five different print formats. The icon for each of these formats is shown to the left of the description.


The **Icon, Description, Panel and Diagram** format (the default) prints the icon, VI description, front panel and block diagram.


The **Using the Panel** format prints the panel, VI description, and the control names and their descriptions.


The **Using as a SubVI** format prints the icon, connector, VI description, and the terminals (data types), names, and descriptions of the connected controls. This format is similar to the LabVIEW function reference format.


The **Complete Documentation** format prints everything, including the icon, connector, description, front panel, information about all front panel controls, the block diagram, and a list of the names of the subVIs.


The **Custom** format prints using the current custom settings. You can change the custom settings by selecting **Configure...** after you have selected the custom option. See the Custom Print Settings topic for information on this dialog box.

Print Documentation Layout Options

The Print Documentation dialog box contains several page layout options that control scaling, page breaks, and headers for the printout. These are described as follows.

Scale Front Panel to Fit and Scale Block Diagram to Fit--These two options control whether the panel and diagram sections are scaled to fit. These options cause LabVIEW to scale the panel and/or diagram down to no less than one-fourth the original size to fit on the fewest number of pages possible.

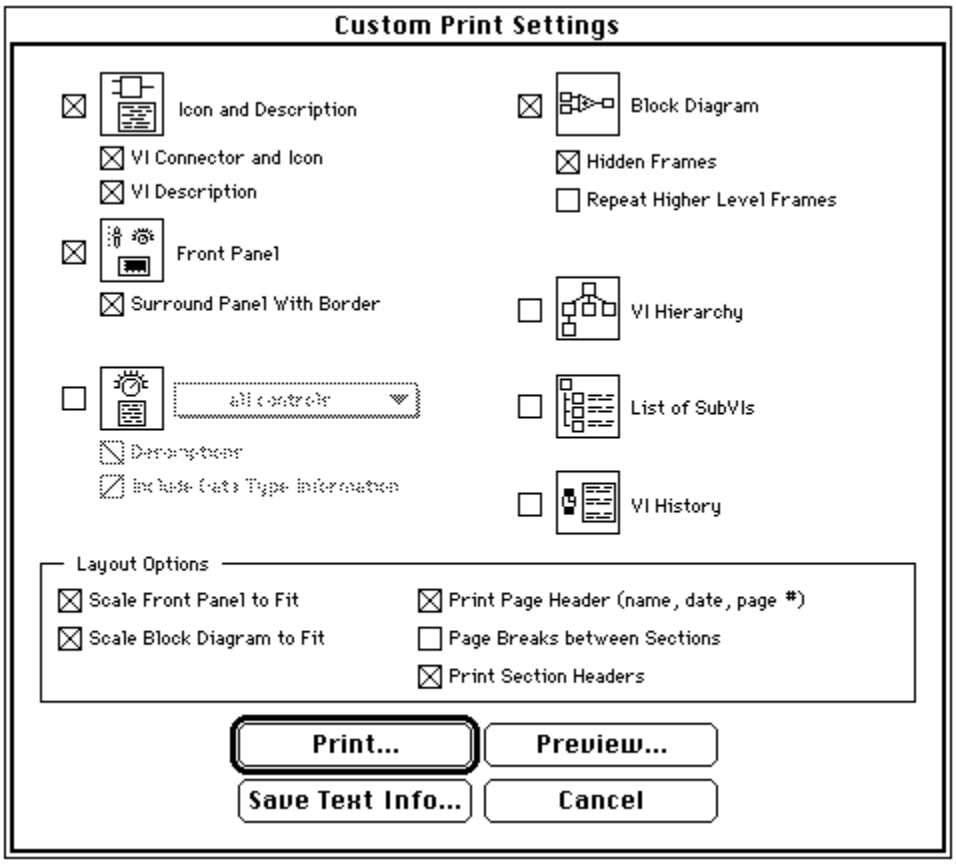
Print Header--Prints a header at the top of every page. This header includes the page number, the VI name, and the last modification date of the VI.

Page Breaks Between Sections--Inserts a page break between the following sections.

- Connector icon and description
- Front panel
- List of front panel control details
- Block diagram
- Block diagram details
- VI hierarchy
- List of subVIs

Custom Print Settings

To bring up the Custom Print Settings dialog box, select the **Custom** button in the Print Documentation dialog box. After you select **Custom**, the **Configure** button is activated. Click on this button to get the dialog box shown in the following illustration.



This dialog box displays the categories of items that can be printed. These categories are listed in the order that they appear on the printout. In addition, the page layout options in this dialog box control scaling, page breaks, and headers for the printout.
The icon for each of the custom print setting options is shown to the left of the description.

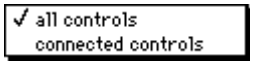
Icon and Description

VI Connector and Icon--Prints a picture of the VI icon along with its inputs and outputs.

VI Description--Prints the VI description.

Front Panel--Prints the front panel.

Controls--Prints a list of the names of the controls and indicators. When an array, cluster, or refnum is encountered, the subcontrols are printed. You can select the following options from the pop-up menu next to this option.



Descriptions--Prints the descriptions next to the control names.

Include Data Type Information--If you print to the printer, then the terminal for each control is printed to the left of the control name. If you save to a text file, the data type is printed after the description.

Block Diagram--Prints the block diagram.

Hidden Frames--Prints the nonvisible frames of Case Structures and Sequence Structures.

Repeat Higher Level Frames--When printing nonvisible frames, prints the visible ones again, in sequence.

VI Hierarchy--Prints a description of the current VI hierarchy in memory, with lines showing connections between VIs and their subVIs. The current VI is highlighted with a box.

List of SubVIs--Prints the icon, name, and path of all subVIs used.

VI History--Prints the history information, if any, for the current VI.

In addition to the layout options found in the Print Documentation dialog box, the Custom Print Settings dialog box has a **Print Section Headers** option.

Print Section Headers--Prints a header for each section (for example, a heading such as “List of SubVIs” before the subVI information). When printing using **Print Documentation**, this selection is automatically set for each format.

Programmatic Printing

If you want to print something under the control of your VI, rather than interactively as with the **Print Window** and **Print Documentation** dialog boxes, use programmatic printing. This type of printing allows you to program your VI to print after every execution. You can also program it to print out the contents of a specific panel or a specific control during execution--perhaps in response to the user pressing a **Print** button.

Programmatic printing allows you to control when printouts occur, as well as the appearance of your printouts, such as whether or not a header is printed.

Enabling Programmatic Printing
Controlling When Printouts Occur
Enhancing Printouts with Transparency, Decorations, and Bitmapped Graphics
Setting up the Page Layout

Enabling Programmatic Printing

To enable programmatic printing, select **Operate»Print at Completion**.

When this option is on, LabVIEW prints the contents of that panel any time the VI completes execution. If the VI is a subVI, LabVIEW prints when that subVI finishes execution, before it returns to the caller.

Controlling When Printouts Occur

In some cases, you do not want a VI to print out every time. You may want it to occur only if the user presses a button, or if some condition occurs, such as a test failure. You may also probably want more control over the format for your printout. Or you may want to print only a subset of the controls, or even just one specific control.

You can create a subVI whose panel is formatted the way you want the VI to look. Instead of selecting **Operate»Print at Completion** on your VI, select it from the subVI. When you want a printout, you can call the subVI, passing the data to it to be printed out.

With this organization, you have complete control over the appearance of the printout, using LabVIEW controls and tools to create very simple or complex printouts.

Enhancing Printouts with Transparency, Decorations, and Bitmapped Graphics

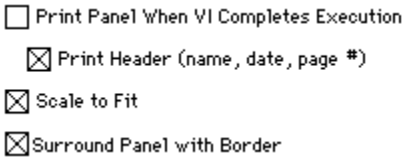
Using transparency and the Color tool, you can hide portions of controls that you do not want to have visible in your printouts. For example, you can use transparency to simplify the appearance of the edges of controls.

Use graphical objects from the **Decorations** menu to highlight sections of your printout. For example, you might surround a section of controls with a box to set it off from the remainder of your panel.

You can also use bitmapped graphics to customize your printout, adding elements such as company logos to your report.

Setting up the Page Layout

The settings in **Edit»Preferences** and in **Printer Setup** (**Page Setup** for the Macintosh and UNIX) affect the appearance of your printouts. You can also **use VI Setup...»Execution Options** to enable or disable some layout options that affect the appearance of your printout. These page layout options, shown in the following illustration, also affect the behavior of the Print window.



The **Print Panel When VI Completes Execution** option is another way to turn on programmatic printing.
If you select **Print Header**, a header including the VI name, the last modification date, and the page number appears at the top of each page.
If you select **Scale to Fit**, and the panel is bigger than a single page, the printout scales down to as little as one-fourth the original size in order to fit the panel on as few pages as possible.
If you select **Surround Panel with Border**, LabVIEW prints a box around the panel.

Other Methods for Printing

You may find that the LabVIEW method of printing is not appropriate for your application. There are some additional techniques that address various printing concerns.

For example, the LabVIEW method of printing is geared towards printing an entire page of data. In some applications, you may prefer to print data on a line-by-line basis. If you have a line-based printer connected to your serial port, you can use the Serial Port VIs to send text to the printer. Doing this generally requires some knowledge of the printer's command language, but has worked well for a number of applications developed by LabVIEW users.

If LabVIEW cannot print data the way you want it, you might consider using another application to print your data by saving the data to a file and then printing from the other application.

(Windows) If you need to print under the control of your VI, you can use `System Exec.vi`, located in the **Functions»Communication** palette, to launch another application. Many applications feature command line options that direct them to print when run. In the case of Note Pad (which is included with all versions of Windows), you can use the following string as the command you pass to `System Exec.vi`.

```
notepad.exe /p filepath
```

With some applications, you may be able to use dynamic data exchange (DDE) to print a document. See the documentation for the individual application to find out whether it responds to any DDE printing commands.

(Macintosh) If you need to print under the control of your VI, you can use the `AESend Print Document.vi` (in **Functions» Communication»AppleEvent**) to direct the other application to print a document. For this to work, the other application must be Apple Event aware (see the other application's documentation for information on whether it responds to Apple Events). See the AESend Print Document VI for a description of how this VI works.

(UNIX) If you need to print under the control of your VI, you can use the `System Exec.vi` (under the **Functions»Communication** palette) to execute a system print command, such as `lpr` or `lp`. Use the UNIX `man lpr` or `man lp` commands to get information on these commands.

Documenting VIs with the Show VI Info... Option

Selecting **Show VI Info...** from the **Windows** menu displays the information dialog box for the current VI. You can use the information dialog box to perform the following functions.

- Enter a description of the VI. The description window has a scrollbar so you can edit or view lengthy descriptions.
- Lock or unlock the VI. You can execute but not edit a locked VI.
- See a list of changes made to the VI since you last saved it.
- View the path of the VI.
- See how much memory the VI uses. The **Size** portion of the information box displays the disk and system memory used by the VI. (This figure applies only to the amount of memory the VI is using and does not reflect the memory used by any of its subVIs.)

  The memory usage is divided into space required for the front panel and the block diagram, VI code, and data space. The memory usage can vary widely, especially as you edit and execute the VI. The block diagram usually requires the most memory. When you are not editing the diagram, save the VI and close the block diagram window to free space for more VIs. Saving and closing subVI panels also frees memory.

Data Storage Formats

This topic discusses the formats in which LabVIEW saves data. This information is most useful to advanced LabVIEW users, such as those using code interface nodes (CINs) and those reading from or writing to files used by the file I/O functions. This topic explains how data is stored in memory, the relationship of type descriptors to data storage, and the method by which data is flattened for file storage on disk.

Data Formats of LabVIEW Front Panel Controls and Indicators
Type Descriptors
Flattened Data

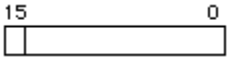
Data Formats of LabVIEW Front Panel Controls and Indicators

The following topics describe the memory representation of LabVIEW front panel controls and indicators.

Boolean Data Formats
Numeric Data Formats
Array Data Storage
String Data Storage
Path Data Storage
Cluster Data Storage

Boolean Data Formats

Booleans are 16-bit integers. The most significant bit contains the Boolean value. If bit 15 is set to 1, then the value of the control is TRUE; if it is set to 0, then the value is FALSE.



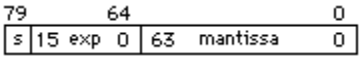
Numeric Data Formats

Extended Data Formats
Double Data Formats
Single Data Formats
Long Integer Data Formats
Word Integer Data Formats
Byte Integer Data Formats

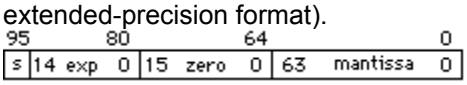
Extended Data Formats

When extended-precision numbers are saved to disk, LabVIEW stores them in a platform-independent 16-bit format. In memory, the size and precision varies depending on the platform.

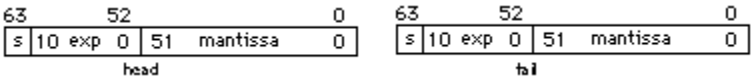
(Windows) Extended-precision floating-point numbers have 80-bit format (80287 extended-precision format).



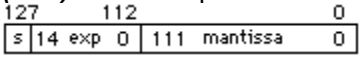
(68K Macintosh) Extended-precision floating-point numbers have 96-bit format (MC68881-MC68882 extended-precision format).



(Power Macintosh) Extended-precision floating-point numbers are stored as two double-precision floating-point numbers combined, i.e., Apple's double-double format. For more details, see Inside Macintosh PowerPC Numerics.



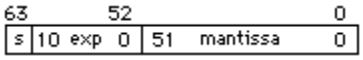
(Sun) Extended-precision floating-point numbers have 128-bit format.



(HP-UX) Extended-precision floating-point numbers are stored the same as double-precision floating-point numbers, as shown in the next illustration.

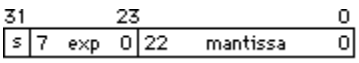
Double Data Formats

Double-precision floating-point numbers have 64-bit IEEE double-precision format (LabVIEW default).



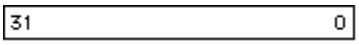
Single Data Formats

Single-precision floating-point numbers have 32-bit IEEE single-precision format.



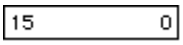
Long Integer Data Formats

Long integer numbers have 32-bit format, signed or unsigned.



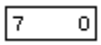
Word Integer Data Formats

Word integer numbers have 16-bit format, signed or unsigned.



Byte Integer Data Formats

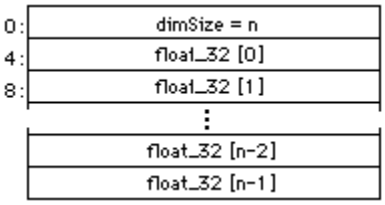
Byte integer numbers have 8-bit format, signed or unsigned.



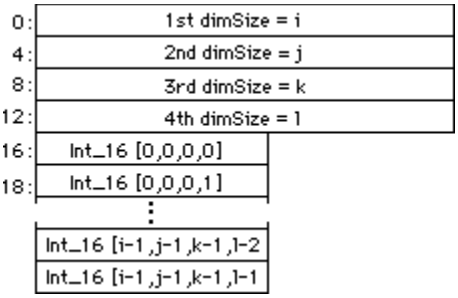
Array Data Storage

LabVIEW stores the size of each dimension of an array in long integers, followed by the data. Because of alignment constraints of certain platforms, the dimension size may be followed by a few bytes of padding so that the first element of the data is correctly aligned.

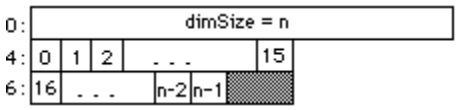
See the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *Code Interface Reference Manual* for further information. The example that follows shows a one-dimensional array of single-precision floating-point numbers. The decimal numbers to the left represent the offsets of locations in memory from which the array begins.



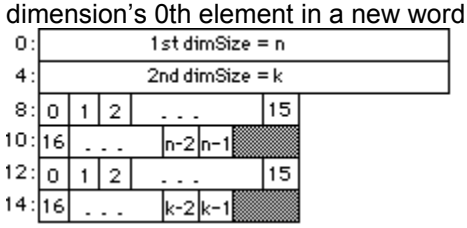
This illustration shows a four-dimensional array of word integers.



LabVIEW stores Boolean arrays differently from Boolean scalars. LabVIEW stores an array of Booleans as packed bits. The dimension size for Boolean arrays is expressed in bits instead of bytes. LabVIEW stores the 0th bit in the highest order bit of its memory word (2^{15}), and the 15th bit in the lowest order bit of its memory word (2^{0}).

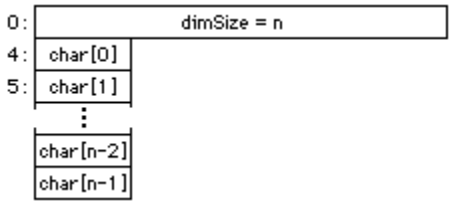


This illustration shows an example of a two-dimensional Boolean array. LabVIEW stores each dimension's 0th element in a new word integer and ignores unused bits from previous dimensions.



String Data Storage

LabVIEW stores strings as if they were one-dimensional arrays of byte integers (8-bit characters).



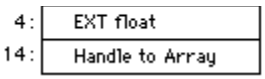
Path Data Storage

LabVIEW stores the path type and number of path components in word integers, followed immediately by the path components. The path type is 0 for an absolute path and 1 for a relative path. Any other value of path type indicates that the path is invalid. Each path component is a Pascal string (P-string) in which the first byte is the length, in bytes, of the P-string (not including the length byte).

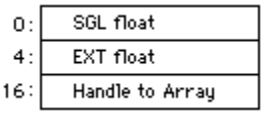
Cluster Data Storage

A cluster stores elements of varying data types according to the *cluster order*. LabVIEW stores scalar data directly in the cluster. LabVIEW stores arrays, strings, handles, and paths indirectly. The cluster stores a handle that points to the memory area in which LabVIEW has actually stored the data. Because of alignment constraints of certain platforms, the dimension size may be followed by a few bytes of padding so that the first element of the data is correctly aligned. See the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *Code Interface Reference Manual* for further information.

The illustrations that follow show a cluster containing a single-precision floating-point number, an extended-precision floating-point number, and a handle to a one-dimensional array of unsigned word integers, presented in that order. See Cluster Controls and Indicators for more information on clusters.



- ***(Macintosh)***



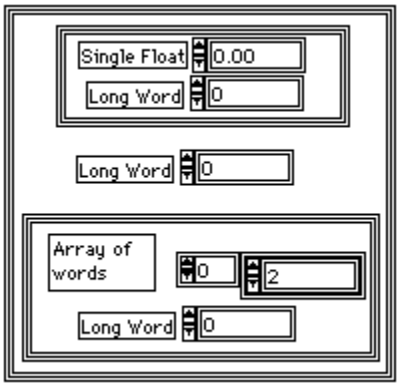
- ***(Sun)***

0:	SGL float
4:	Filler
8:	EXT float
24:	Handle to Array

- ***(HP-UX)***

0:	SGL float
4:	Filler
8:	EXT float
16:	Handle to Array

In the next example, LabVIEW does not store the embedded clusters indirectly. LabVIEW stores the data inside the embedded clusters directly, as if the data were not embedded in the subcluster. LabVIEW stores only arrays, strings and handles indirectly.



0:	32-bit float
4:	32-bit float
8:	32-bit int
12:	Handle to Array
16:	32-bit int

Type Descriptors

Each wire and terminal in the LabVIEW block diagram has a data type associated with it. LabVIEW keeps track of this type with a structure in memory called a *type descriptor*. This descriptor is a string of word integers that can describe any data type in LabVIEW. Numeric values are given in hexadecimal, unless otherwise noted.

The generic format of a type descriptor is: `<length> <typecode>`.

Some type descriptors have additional information following the type code. Arrays and clusters are structured or aggregate data types because they include other types. For example, the cluster type contains additional information about the type of each of its elements.

The first word (16 bits) in any type descriptor is the length, in bytes, of that type descriptor (including the length word). LabVIEW reserves the high-order byte of the type code for internal use. When comparing two type descriptors for equality, you should ignore this byte; two descriptors are equal even if the high-order bytes of the type codes are not.

The type code encodes the actual type information, such as single-precision or extended-precision floating-point number, as listed in the following table. These type code values may change in future versions of LabVIEW. The *xx* in the type descriptor columns represent reserved LabVIEW values and should be ignored.

The following tables list scalar numeric and non-numeric data types, the type codes, and type descriptors.

Scalar Numeric Data Types Table

Data Type	Type Code (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Byte Integer	01	0004 *xx*01
Word Integer	02	0004 *xx*02
Long Integer	03	0004 *xx*03
Unsigned Byte Integer	05	0004 *xx*05
Unsigned Word Integer	06	0004 *xx*06
Unsigned Long Integer	07	0004 *xx*07
Single-Precision Floating-Point Number	09	0004 *xx*09
Double-Precision Floating-Point Number	0A	0004 *xx*0A
Extended-Precision Floating-Point Number	0B	0004 *xx*0B
Single-Precision Complex Floating-Point Number	0C	0004 *xx*0C

Double-Precision Complex Floating-Point Number	0D	0004 *xx*0D
Extended-Precision Complex Floating-Point Number	0E	0004 *xx*0E
Enumerated Byte Integer	15	<nn> *xx*15 <k> <k pstrs>
Enumerated Word Integer	16	<nn> *xx*16 <k> <k pstrs>
Enumerated Long Integer	17	<nn> *xx*17 <k> <k pstrs>
Single-Precision Physical Quantity	19	<nn> *xx*19 <k> <k base-exp>
Double-Precision Physical Quantity	1A	<nn> *xx*1A <k> <k base-exp>
Extended-Precision Physical Quantity	1B	<nn> *xx*1B <k> <k base-exp>
Single-Precision Complex Physical Quantity	1C	<nn> *xx*1C <k> <k base-exp>
Double-Precision Complex Physical Quantity	1D	<nn> *xx*1D <k> <k base-exp>
Extended-Precision Complex Physical Quantity	1E	<nn> *xx*1E <k> <k base-exp>

Non-Numeric Data Types Table

Data Type	Type Code	Type Descriptor
Boolean	20	0004 *xx*20
String	30	0004 *xx*30
Handle	31	0006 *xx*31 <kind>
Path	32	0004 *xx*32
Pict	33	0004 *xx*33
Array	40	<nn> 0x40 <k> <k dims> <element type descriptor>
Cluster	0x50	<nn> 0x50 <k> <k element type descriptors>

The minimum value in the size field of a type descriptor is 4 (as shown in the first table). However, any type descriptor can have a name appended (a P-string) in which case the size is larger by the length of the name.

Notice that the array and cluster data types each have their own type code. They also contain additional information about their dimensionality (for arrays) or number of elements (for clusters), as well as information about the data types of their elements.

In the following example of an enumerated type for the items am, fm, fm stereo, each group represents a 16-bit word. The space enclosed in quotes (" ") represents an ASCII space.

```
0016 0015 0003 02a m02 fm 09f m" "st er eo
```

0016 indicates 22 bytes total. 0015 indicates an enumerated byte. 0003 indicates that there are three items.

In the following example of a physical quantity for the double-precision width units m / s each group represents a 16-bit word.

```
000E 001A 0002 0002 FFFF 0003 0001
```

000E indicates 14 bytes total. 0x1A indicates that these are double-precision width units. 0002 indicates two base-exponent pairs. 0002 denotes the seconds base index. FFFF is the exponent of seconds. 0003 denotes the meters base index. 0001 is the exponent of meters.

Note: All physical quantities are stored internally in terms of base units, regardless of what unit they are displayed as. Table 9-2, Base Units, shows the nine bases which are represented by indices 0-8 for radians-candela.

Array Type Descriptors
Cluster Type Descriptors

Array Type Descriptors

The type code for an array is 0x40. Immediately after the type code is a word containing the number of dimensions of the array. Then, for each dimension, an unsigned long integer contains the size, in elements, of that dimension. Finally, after each of the dimension sizes, the type descriptor for the element appears. The element type may be any type except an array. The dimension size for any dimension may be FFFFFFFF (-1). This means that the array dimension size is variable. Currently, all arrays in LabVIEW are variable-sized. LabVIEW stores the actual dimension size with the data. The dimension size is always greater than or equal to zero. The following is a type descriptor for a one-dimensional array of double-precision floating-point numbers:

```
000E 0040 0001 FFFF FFFF 0004 000A
```

000E is the length of the entire type descriptor, including the element type descriptor. The array is variable-sized, so the dimension size is FFFFFFFF. Notice that the element type descriptor (0004 000A) appears exactly as it does for a scalar of the same type.

The following is an example of a type descriptor for a two-dimensional array of Booleans.

```
0012 0040 0002 FFFF FFFF FFFF FFFF 0004 0020
```

Cluster Type Descriptors

The type code for a cluster is 0x50. Immediately after the type code is a word containing the number of items in the cluster. After this word is the type descriptor for each element in *cluster order*. For example, consider a cluster of two integers: a signed-word integer and an unsigned long integer:

```
000E 0050 0002 0004 0002 0004 0007
```

`000E` is the length of the type descriptor including the element type descriptors.

The following is a type descriptor for a multiplot graph (the numeric types can vary):

`0028 0040 0001 FFFF FFFF`	1D array of
`001E 0050 0001`	...1 component cluster of
`0018 0040 0001 FFFF FFFF`	...1D array of
`000E 0050 0002`	...2 component cluster of
`0004 000A`	...double-precision floating-point number
`0004 0003`	...long integer

Flattened Data

Two LabVIEW internal functions convert data from the LabVIEW memory storage format to a form more suitable for writing to or reading from a file.

String, Handle, and Path Flattened Data
Array Flattened Data
Cluster Flattened Data

Because strings and arrays are stored in handle blocks, clusters containing these types are *discontiguous*. In general, data in LabVIEW is stored in *tree* form. For example, a cluster of a double-precision floating-point number and a string is stored as an 8-byte floating-point number, followed by a 4-byte handle to the string. The string data is not stored adjacent in memory to the extended-precision floating-point number. Therefore, if you want to write the cluster data to disk, you have to get the data from two different places. Of course, with an arbitrarily complex data type, the data may be stored in many different places.

When LabVIEW saves data to a VI file or a datalog file, it *flattens* the data into a single string before saving it. This way, even the data from an arbitrarily complex cluster is made contiguous, instead of being stored in several pieces. When LabVIEW loads such a file from disk, it must perform the reverse operation—it must read a single string and unflatten it into its internal LabVIEW form.

LabVIEW normalizes the flattened data to a standard form so that the data can be used unaltered by VIs running on any platform. It stores numeric data in big endian form (most-significant byte format), and it stores extended precision floating-point numbers as 16-byte quantities using the Sun extended-precision format described earlier in this section. Similar transformations may be necessary when reading data written by an application other than LabVIEW.

Note: When writing data to a file for use by an application other than LabVIEW, you may need to transform your data after flattening it. Other Windows applications typically expect numeric data to be in little endian form (least-significant byte first). Other Windows and Macintosh applications typically expect extended-precision floating-point numbers to be in the 80-bit and 96-bit formats described previously, respectively.

The Flatten to String and Unflatten from String block diagram functions use the internal flatten and

unflatten functions.

Scalar Flattened Data

The flattened form of any numeric type, as well as the Boolean type, contains only the data in big endian format. For example, a long integer with value -19 would be encoded as FFFF FFED. A double-precision floating-point number with a value approximately equal to pi is 0X 40 09 21 FB 54 44 2D 18. A Boolean True is 8xxx, 9xxx, ..., Exxx, or Fxxx.

The LabVIEW file form for extended-precision numbers is based on the SPARC quadruple-precision form, which is a 16-bit biased exponent, followed by a mantissa which is 112 bits long. The Macintosh and PC have an explicit bit, the bit left of the binary point. In normalized numbers, this bit is always a one. It is only a zero when the exponent is zero (very small, denormalized numbers). On the SPARC, the bit left of the binary point is an implicit bit (assumed always to be one) which does not appear in the representation.

String, Handle, and Path Flattened Data

Because strings, handles, and paths have variable sizes, the flattened form is preceded by a "normalized" long integer that gives their length in bytes. For paths, this length is preceded by four characters: PTH0. For instance, a string type with value ABC would be flattened to 0000 0003 4142 43.

The flattened format is similar to the format that the string takes in memory. However, handles and paths do not have a length value preceding them when they are stored in memory, so LabVIEW obtains this value from the actual size of the data in memory, and prepends it when the data is flattened.

Array Flattened Data

The data for a flattened array is preceded by "normalized" long integers that give the size, in elements, of each of the dimensions of the arrays. The slowest varying dimension is first, followed in order by the succeeding, faster-varying dimensions, just as the dimension sizes are stored in memory. The data follows immediately after these dimension sizes in the same order in which it is stored in memory. LabVIEW also flattens this data if necessary. The following is an example of a two-dimensional array of six 8-bit integers.

```
{ {1, 2, 3}, {4, 5, 6} } is stored as 0000 0002 0000 0003
0102 0304 0506.
```

The following is an example of a flattened one-dimensional array of Boolean variables:

```
{T, F, T, T} is stored as 0000 0004 B000. (The binary
encoding of "B" is 1011.)
```

Boolean arrays are stored as packed bits unlike other arrays.

Cluster Flattened Data

A flattened cluster is the concatenation (in cluster order) of the flattened data of its elements. So, for example, a flattened cluster of a word integer of value 4 (decimal) and a long integer of value 12 would be 0004 0000 000C.

A flattened cluster of a string ABC and a word integer of value 4 is 0000 0003 4142 4300 04.

A flattened cluster of a word integer of value 7, a cluster of a word integer of value 8, and a word integer of value 9 would be 0007 0008 0009.

To unflatten this data, you just use the reverse process. The flattened form of a piece of data does *not* encode the type of the data; you need the type descriptor for that. The Unflatten From String function requires you to wire a data type as an input, so that the function can decode the string properly.

Charts and Graphs

How do I wire data to a graph or chart?
What is the basic difference between graphs and charts?
How do I clear a chart programmatically?
How do I avoid the graphs flashing each time it is updated?
How do I update a graph without clearing it?
How do I flip the scales on my graph or chart?
How do I make a graph or chart display information in a time format?
How do I make the x-axis of a chart display real time?
How can I create a bar graph?
How can I create polar plots and Smith charts?
How can I make a label for the y-axis that is rotated 90 degrees?
How do I place a text label on a cursor?
After Ive added a cursor to my graph, how do I remove it?
Where is the Clear button that was available on the Graph and Chart palettes in LabVIEW 3.0.x?

How do I wire data to a graph or chart?

Show the Help window and move the Wiring tool across a graph terminal in the block diagram to see a brief description of how to wire basic graph types. There are excellent examples on graphs and charts in the `examples\general\graphs` directory.

What is the basic difference between graphs and charts?

Graphs and charts differ in the way they display and update data. VIs with graphs usually collect the data in an array and then plot it on the graph, similar to a spreadsheet that first stores the data then generates a plot of it. In contrast, a chart appends new data points to those already in the display. In this manner, you can see the current reading or measurement in context with data previously acquired. The length of the chart history buffer can be set by a pop-up option on the chart. Of course, it is possible to implement a history buffer with the graph indicators as well; however, this needs to be done in the block diagram. These features are already built into the chart.

How do I clear a chart programmatically?

Wire an empty array to the History Data attribute. The data type of this empty array is the same as the data type wired to the chart. There is an excellent example which illustrates this technique in `examples\general\graphs\charts.llb\How to Clear Charts & Graphs.vi`.

How do I avoid the graphs flashing each time it is updated?

Use the **Smooth Updates** option in LabVIEW to keep graphs from flashing. This option is located in the **Data Operations** submenu of the graph pop-up menu. Using **Smooth Updates** means that LabVIEW will first redraw graphs to an off-screen buffer before copying the image to the display. While producing a smoother update to the indicators, this option is generally slower and requires more memory.

Smooth Updates can be set globally in LabVIEW through the **Edit»Preferences»Front Panel** dialog box. It can be turned on or off for individual graphs from their pop-up menus.

How do I update a graph without clearing it?

All of the graphs (waveform, XY, and intensity) always clear before writing new data; however, it is a simple procedure to keep track of the data previously written to the graph and append new data with each write. The `examples\general\arrays.llb\Separate Array Values.vi` demonstrates the technique of using the Build Array function to append new values to an array. This VI is explained in detail in Chapter 4, *Arrays, Clusters, and Graphs*, of the *LabVIEW Tutorial Manual*.

How do I flip the scales on my graph or chart?

To flip the scales for the x-axis or y-axis, use the attribute X Flipped or Y Flipped. If the X Flipped attribute is set to TRUE, then the minimum of the x-axis scale is set to the right and the maximum to the left. Similarly, if the Y Flipped attribute is set to TRUE, the minimum of the y-axis scale is at the top and the maximum is at the bottom.

How do I make a graph or chart display information in a time format?

LabVIEW can display numeric information in a graph or chart in a relative time format. Use **the X Scale»Formatting...** or **Y Scale»Formatting...** pop-up option to set the x-axis or y-axis scale format to **Relative Time**.

How do I make the x-axis of a chart display real time?

See the example VI called `Real-Time Chart.vi`, located in `examples\general\graphs\charts.llb`. It unbundles the date time cluster from the Seconds to Date/Time function into hours, minutes, and seconds. It then converts these numbers into the number of seconds that have elapsed since midnight. Finally, this number is used as an input to the Xo and delta X attribute of a chart that has its x-scale formatting set to **Relative Time**. Or you can use the **Absolute Time Format & Precision** option.

How can I create a bar graph?

See the examples in `examples\general\graphs\ bargraph.llb`. In this library you will find an example bar graph and two VIs you can use to create bar graphs out of an array input. The LabVIEW Picture Control Toolkit can also create bar graphs.

How can I create polar plots and Smith charts?

The LabVIEW Picture Control Toolkit contains examples with routines to create polar plots and Smith charts. The toolkit is a versatile graphics package for creating arbitrary front panel displays. The Picture VI Library, which ships with the toolkit, implements a common set of drawing commands for building the graphic images.

How can I make a label for the y-axis that is rotated 90 degrees?

LabVIEW does not have the capability to rotate text. To use rotated text in LabVIEW, first create it in an application that has this capability, such as Windows Paintbrush, and save the graphic in a format that LabVIEW can import (see the Importing Graphics from Other Programs topic for more information on importing graphics into LabVIEW). You can then import this graphic into LabVIEW and place it on the y-axis of the graph or chart.

How do I place a text label on a cursor?

You must first use the Cursor Name Visible attribute to display the cursor name. You can then use the

Cursor Name attribute to assign a name to the cursor or edit the name in the graphs cursor display on the front panel.

After Ive added a cursor to my graph, how do I remove it?

You must empty the array that contains the clusters holding each cursors information. Select Data Operations»Empty Array from the cursor displays pop-up menu, or write an empty array to the Cursor List Attribute Node programmatically.

Where is the Clear button that was available on the Graph and Chart palettes in LabVIEW 3.0.x?

The Clear button is no longer available on the Graph and Chart palettes. To clear the chart or graph, use the pop-up menu and select **Data Operations**»**Clear Graph/Chart**.

Error Messages and Crashes

All Platforms

A dialog box appears stating that LabVIEW is out of memory.

While running LabVIEW, I receive a LabVIEW failure message which includes a source code file and line number.

Windows Only

LabVIEW crashes when printing.

LabVIEW tends to crash randomly. The crashes may be general protection faults or LabVIEW failure messages which include a source code file and line number.

A dialog box appears stating that LabVIEW is out of memory.

LabVIEW allocates memory as needed, but arrays and strings must be stored in a contiguous block of memory. If LabVIEW is unable to find a block of unused memory (physical or virtual) that is large enough for the string or array, a dialog box appears to indicate that LabVIEW was not able to allocate the required memory.

If you want to save changes to your VIs after LabVIEW runs out of memory, you might save your VIs to another location or make sure that you have a backup copy. After you restart LabVIEW with more memory, you can load these VIs into memory, check to see that the save was successful, and proceed in your VI development.

(Windows) LabVIEW crashes when printing.

This crash may be related to the video driver. See the (Windows) LabVIEW tends to crash randomly topic for more information. The crashes may be general protection faults or LabVIEW failure messages which include a source code file and line number.

While running LabVIEW, I receive a LabVIEW failure message which includes a source code file and line number.

This indicates that an error has occurred and LabVIEW cannot proceed. Please notify National Instruments of this message and explain what action triggered the message.

(Windows) LabVIEW tends to crash randomly. The crashes may be general protection faults or LabVIEW failure messages which include a source code file and line number.

Often random crashes involve problems with the video driver on the machine. To establish if this is the case, use standard VGA as the video driver. To change video drivers, select VGA from the list of video drivers in Windows Setup. LabVIEW uses the standard Windows API for its graphical calls; however, we have found that many video drivers do not fully conform to this standard. If the crashes do not occur with the standard VGA driver, then you should get an update to your video driver. Both your PCs manufacturer and the manufacturer of the video card should be able to provide you with the latest release of the video driver software. Several customers have reported success after updating their drivers, particularly when using the Mach32 (shipped with ATI boards and Gateway machines) or Cirrus Logic video drivers.

If the situation is not resolved by updating the video driver, contact National Instruments.

(Windows) While running LabVIEW, I receive a memory parity error, followed by a General Protection Fault.

Memory parity errors are the result of bad memory in your machine. This memory might be virtual memory, indicating a problem on your hard disk, or physical memory, indicating a bad SIMM. To check for problems with your hard disk, use a standard disk utility package such as Norton Utilities. To check for problems with physical RAM, you need to physically rotate the SIMMs in your machine, rebooting the PC after each rotation. When the bad SIMM is placed in the lowest bank of memory, the machine will not boot at all. The SIMM should be replaced.

Miscellaneous

All Platforms

What is info-labview and how do I subscribe?
How can I dynamically load and run VIs?
How do I replace a subVI with another VI that has the same name?
Loading two VIs that call two distinct subVIs with the same name causes problems.
When loading and executing multiple VIs, how is the priority of execution assigned to the VIs?
How can LabVIEW service interrupts without using polling?
With nested While Loops, how do I stop both loops without anything in the outer loop executing after the inner loop stops?
What is Panel Order?
How can I get LabVIEW to automatically launch a particular VI?
How do I programmatically change the entries of an enumerated type?
How do I hide the menu bars of a LabVIEW VI?
How do I disable mouse interrupts to improve performance?

What is info-labview and how do I subscribe?

`Info-labview` is a user-sponsored and supported network of LabVIEW users. LabVIEW users can post information to a specific e-mail address (see below); the posting is then broadcast to all users subscribing to the list. Other users may then respond back to the group or perhaps directly to the individual who wrote the post.

Several power users, as well as some National Instruments employees, subscribe to the list and respond to various issues. Overall the Info LabVIEW group has been received extremely well. Although the group is not connected with National Instruments, official responses are made on occasion to the group. Most of the time the conversations take place strictly between LabVIEW users.

Users who want to subscribe to the list should send e-mail to this address:

`info-labview-request@pica.army.mil`

There are two ways for users to receive the postings--the standard format and digest format. Standard format means you receive the postings as they occur. Digest format means you receive one package at the end of each day containing all postings for the day. Subscribers should specify digest format in the e-mail messages they send to info-labview-request if they wish to receive one bulk message daily.

Once you subscribe to the list, you will receive postings from other subscribers. Traffic in the group is fairly active, from 10 to 30 postings per day. To post information to the list, send e-mail to this address:

`info-labview@pica.army.mil`

How can I dynamically load and run VIs?

When a subVI exists in the block diagram of another VI, the subVI is loaded into memory as soon as the calling VI is loaded into memory. For memory considerations, you may want LabVIEW to dynamically load and unload VIs from memory during the execution of your program. Special VI Control VIs allow you to do this. These are located in the **Advanced**»**VI Control** palette. For more information on these VIs, see the

VI Control VIs topic.

(Macintosh) In addition to the platform-independent VI Control VIs, you can use Apple Events on the Macintosh to dynamically load and run VIs. The VIs are located in the palette **Functions**»**Communication**»**AppleEvent** or its subpalette **LabVIEW Specific Apple Events**, and are called:

- AESend Finder Open
- AESend Open, Run, Close VI
- AESend Run VI
- AESend Close VI

How do I replace a subVI with another VI that has the same name?

LabVIEW references all VIs by name, and thus never loads two VIs with the same name, regardless of the paths to those VIs. This includes subVIs. When you select **Replace** on a subVI and ask LabVIEW to replace it with a VI of the same name, LabVIEW realizes it already has a copy of the VI in memory, and replaces the VI with itself.

To load the new copy of the subVI, you must first remove the old copy from memory. The VIs currently are listed in the **Windows** menu. The easiest way to remove the subVI from memory is to close the subVI and any other VIs that call the subVI. Open the new copy of the subVI into memory, check **File**»**Get Info...** to confirm that this is the version you want, and then re-open the main VI. When LabVIEW tries to link in the subVI, it will find a copy in memory (the new version) and will use it rather than search for the previous version.

The solution to many of these problems is always to give subVIs unique names.

Loading two VIs that call two distinct subVIs with the same name causes problems.

For example, `main1.vi` and `main2.vi` both call distinct subVIs which are both named `subVI.vi`.

1. `main1.vi` is loaded into memory, `subVI.vi` is loaded in because it is called by `main1.vi`.
2. `main2.vi` is loaded into memory, linker information tells LabVIEW to load `subVI.vi`. LabVIEW realizes that it already has subVI.vi loaded into memory, tries to link `main2.vi` with this `subVI.vi`. If the connector pane is exactly the same, it will probably link fine (but the behavior may be very peculiar). If the connector pane is different, the user will get a bad linkage error.

Solution: Always give subVIs unique names.

When loading and executing multiple VIs, how is the priority of execution assigned to the VIs?

The LabVIEW execution system allows you to execute multiple VIs or subdiagrams in parallel and assign priorities to VIs. Priorities range from 0 (low) to 3 (high), with an additional, highest, level of subroutine. At any one time only one section of code executes; other sections wait on an execution queue. If the code does not complete in a given time period, LabVIEW puts it back on the queue, and the frontmost queue element starts running. In this fashion, execution passes round-robin between VIs or subdiagrams.

You can alter the priority of a VI through the dialog box in **VI Setup**. Notice that a VI with high priority automatically bumps its subVIs to the same level of priority. For example, if `main.vi` has priority 3 (high) and `subVI.vi` has priority 1 (not as high), then when main.vi calls `subVI.vi`, `subVI.vi` has priority 3.

With priorities, the LabVIEW internal execution queue has multiple entry points. The queue is always ordered via priority, as in this example:

```
sss322211111
```

If a new item with priority 3 is placed on the queue, then the queue would look like the following:

```
sss3322211111
```

Notice that the new 3 item gets placed behind all s items (compiled as subroutines) and other priority 3 items.

A potential danger in using VI priority is that low priority items will never execute if higher level tasks are always available to run. Also, priority does not specify which of two items in parallel gets executed first. If something with high priority is put on the queue, it will execute as soon as the currently executing lower priority node checks the queue.

How can LabVIEW service interrupts without using polling?

Occurrences are a method for LabVIEW to make a section of code wait for a particular event. For example, you can write a CIN that spawns a process (in Windows through a DLL, for example) that will generate a LabVIEW occurrence whenever some event takes place in the DLL. Some section of LV code is executing, and another section or VI is waiting for the event. When the event happens, it logs the occurrence in LabVIEW, and the appropriate code will begin to execute... as soon as its turn on the queue comes up.

Internally, LabVIEW looks at the queue whenever it finishes a block of atomic code. This means that if something else is executing it will complete before the occurrence, and hence the interrupt, is handled. Notice that there is no definite time limit on how long this might be, particularly if the VI contains CINs, which may take a very long time to complete execution.

With nested While Loops, how do I stop both loops without anything in the outer loop executing after the inner loop stops?

Put the code you do not want to execute on the last iteration into a Case Structure. The condition terminal of the inner loop should be connected to the selection terminal of the Case Structure.

What is Panel Order?

Edit»Panel Order determines the order of the objects on the front panel. In other words, if you were using `<Tab>` on the keyboard to move between objects, the key focus would change from object to object in the order determined by the panel order. Only controls can receive key focus; the `<Tab>` key does not switch focus to indicators. By default, the panel order is the order in which you create objects on the front panel.

With front panel data logging, the panel order determines the order in which the different objects are compacted into a cluster within the file.

How can I get LabVIEW to automatically launch a particular VI?

By default, LabVIEW launches and creates an `Untitled1.vi`. You may want LabVIEW to automatically launch a particular VI. You can also set the VI to run when loaded, so that launching LabVIEW not only opens your VI but also starts running it. Set the appropriate **Execution Options** in **VI Setup...** to make a VI run when opened. If you specify a library `(.llb)` to be opened when LabVIEW is launched, LabVIEW

opens all VIs marked **Auto Load**. Use **File**»**Edit VI Library...** to mark VIs with **Auto Load**. Finally, if you specify a library and no VIs are marked **Auto Load**, a file dialog box prompts the user to select a VI within the specified `.llb`.

To tell LabVIEW to launch a particular VI, follow the instructions that follow for the relevant platform.

(Windows) Select the LabVIEW icon, select **File**»**Properties** (in Program Manager), and change the pathname in **Command Line** to point towards your VI. For example, to make LabVIEW automatically load `test.vi`, set the **Command Line** to:

```
c:\labview\labview.exe test.vi
```

If test.vi is inside a library called `test.llb`, then:

```
c:\labview\labview.exe test.llb\test.vi
```

You may need to specify the full path to the VI.

(Macintosh) There is no way to get the LabVIEW application to automatically launch a particular VI on the Macintosh; however, you should be able to launch a VI by double-clicking on its icon in the Finder. If you have multiple copies of LabVIEW installed on your machine, the Finder will decide which copy is launched when you double-click on the VI (you have no control over which copy is launched). For example, the Finder may decide to launch the Run-Time System if you have both the Run-Time System and Full Development System installed. If you are running System 7, you can use Drag & Drop to launch one or more VIs, or llbs.

(UNIX) LabVIEW responds to command-line options to launch a particular VI when opened. Therefore, you can type:

```
labview /usr/home/test.vi
```

or

```
labview /usr/home/test.llb/test.vi
```

to launch `test.vi`, depending on the correct path. You can use a simple script to make a command that launches LabVIEW with a particular VI.

How do I programmatically change the entries of an enumerated type?

You cannot programmatically change the type (the strings) of an enumerated data type, just as you cannot programmatically change an integer control into a double or a string control into a path control. The strings in an Enum are a part of its data type and thus can only be changed during edit time. It is possible to read the strings of the Enum through an Attribute Node, but you cannot write them using an Attribute Node.

If you need to programmatically change the text values in the Enum, use a text ring control instead. You can use the ring control to programmatically read and write the strings through the Strings[] attribute. Because text rings are just numeric controls, they do not associate the strings with their data type.

How do I hide the menu bars of a LabVIEW VI?

All VI attributes, including whether the menu bar and the toolbar are displayed, are set through the dialog

box in **VI Setup...**. To access **VI Setup...**, pop up on the VI icon in the upper-right corner of the front panel. See Creating Pop-up Panels and Setting Window Features, for more details.

How do I disable mouse interrupts to improve performance?

Interrupts caused by moving or clicking the mouse take CPU time. In some extremely time-critical applications, this interrupt activity can result in a loss of data or some other problem with the application. There is no known way to disable mouse interrupts while an application is running. Make sure that the mouse does not move during the time-critical portions of your application.

Platform Issues and Compatibilities

What is required to transfer VIs between platforms?

Once a VI file is brought to another platform, no conversion is necessary for LabVIEW to read it. When LabVIEW opens the VI, it recognizes that it has been compiled for a different platform, and recompiles it for the new platform. This means that you must include the block diagram of a VI if you want to take it to another platform. For VIs containing CINs, you need to recompile the CINs on the new platform and then relink them to the ported VI. You should eliminate platform-specific functions (for example, Apple Events for Macintosh, DDE for Windows, and so on) before you port a VI to another platform.

You can move VIs between platforms via networks, modems, and disks. LabVIEW VIs are saved in the same file format on all platforms. If you transfer VIs across a network via ftp or modem, make sure that you specify a binary transfer.

If disks are the method of transfer, you need disk conversion utilities to read the disks from other platforms. Conversion utilities change the format of files stored on disk because each platform (Macintosh, Windows, Sun) saves files to disk in a different format. Most file conversion utilities not only read files from another platform, but also write files in the disk format of that platform. For example, there are utilities such as MacDisk and TransferPro available for the PC that transfer Macintosh disks to the PC format and vice versa. On the Macintosh, DOS Mounter and Apple File Exchange are two utilities that convert files on DOS-formatted disks to the Macintosh format and vice versa. For the Sun and HP, there is PC File System (PCFS) that allows SunOS and HP-UX to read and write DOS-formatted disks.

In order to ease porting between platforms, you can save your VIs into a VI library. To facilitate saving all your VIs into a library, you can select the **Development Distribution** option from the Save with Options dialog box. This enables you to save all non-`vi.lib` VIs, controls, and external subroutines to a single library.

Printing

All Platforms

When using automatic (programmatic) printing while executing a VI, why is the information not printed until the VI

How do I print a single control from the front panel (for example, a graph)?

How can I print a string from LabVIEW?

How do I print all of the data on a Waveform Chart?

Why does LabVIEW crash when printing?

How do I use PostScript printing in LabVIEW?

The text on labels and front panel controls is clipped when the VI prints.

When I select PostScript printing in LabVIEW, why does my printout yield a pile up of printed text at the top of the page?

Windows Only

How do I select bitmap printing?

When using automatic (programmatic) printing while executing a VI, why is the information not printed until the VI stops executing?

Programmatic printing by design does not print a VI or subVI until it stops execution. For printing during execution, use a) **File**»**Print Window** to print the front panel manually, or b) a call to a subVI that accepts the input data to its controls (the front panel might be identical to that of the executing VI) and prints its results with programmatic printing. For more details, see the topic Programmatic Printing.

How do I print a single control from the front panel (for example, a graph)?

To print only a graph from the front panel, create a subVI with a graph on its front panel. Change the graph from an indicator to a control. Open the subVI and select **Operate**»**Print at Completion**. Assign the subVI a connector and pass the data from the graph on the main VI to the graph on the subVI. Every time your main VI calls the subVI, it will automatically print the graph.

How can I print a string from LabVIEW?

Use the Serial Port Init.vi in `VI.lib\Instr\ SERIAL.llb` to initialize the port where the printer is connected (`LPT1`, `LPT2`, and so on, on the PC, or the printer port on the Mac) and then the `Serial Port Write.vi` to write the string to the initialized port. The printer will see the data at the port and print it. Doing this generally requires some knowledge of your printers command language, but has worked well for a number of applications developed by LabVIEW users. See the Serial I/O Common Questions topic as well as the Serial Port VI Overview topics for more details.

(Windows and UNIX) Use the System Exec VI to print a file through a command line function. The VI is located in **Functions**»**Communication**»**AppleEvent**. For more details, see the topic Other Methods for Printing. You can also use the programmatic print option.

(Macintosh) You can use the AESend Print Document VI to direct the other application to print a document. The VI is located in **Functions**»**Communication**»**AppleEvent**. For more details, see the topic

Other Methods for Printing. You can also use the programmatic print option.

How do I print all of the data on a Waveform Chart?

The Waveform Chart will only print the data that is displayed when the front panel is printed. To print all of data in a chart, including that in the history buffer, first autoscale the x-axis of the chart by popping up on the chart, showing the chart palette and locking the x-axis, or setting the x-axis programmatically through a chart Attribute Node.

Why does LabVIEW crash when printing?

On Windows, this crash may be related to the video driver.

See the question concerning random crashes on your platform in the topic Error Messages and Crashes for information on how to proceed.

How do I use PostScript printing in LabVIEW?

If your printer supports PostScript language, first install and use the PostScript printer driver for your printer. In LabVIEW, go to **Edit**»**Preferences**»**Printing** and select **PostScript printing**.

If you have a PostScript printer, you can take advantage of the following benefits:

- PostScript printouts reproduce the image of the screen more accurately.
- PostScript facilitates high-resolution graphs.
- PostScript reproduces patterns and line styles more accurately.

The text on labels and front panel controls is clipped when the VI prints.

This occurs when the size of the font for the printer does not match the size of the font on the monitor.

1. If your printer supports PostScript language, use the PostScript printer driver for your printer and you will get higher quality printouts. See the question above for more information on PostScript printing.
2. Enlarge the labels and front panel controls so that, when the text expands because of a font mismatch, the text still fits in the control or label boundary.
3. Find a font which is the same size on both the monitor and the printer. Many printer drivers have font substitution algorithms to assist this process.
4. ***(Windows)*** Use bitmap printing, which will make the printout exactly match what you see on the monitor.

When I select PostScript printing in LabVIEW, why does my printout yield a pile up of printed text at the top of the page?

LabVIEW translates the VI print data into PostScript (.PS) format and sends it to the Windows printer driver as PostScript text. If your printer driver does not support PostScript printing, the printout will be the actual text of the PostScript file instead of a graphics image. See the How do I use PostScript printing in LabVIEW? topic for more information.

(Windows) How do I select bitmap printing?

Go to **Edit**»**Preferences** and select **Printing**. The last option on the screen is Bitmap printing.

About This File

Copyright

Limited Warranty

The software media is warranted against defects in materials and workmanship for a period of two years from the date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

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.

Trademarks

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.

Test Executive

Features

- Test sequencing with dependencies
- Test results logged and ASCII test reports created
- Halt and loop capabilities on individual test failure
- Force conditions
- Continuous UUT testing mode
- User prompts, pass/fail banners, and error reporting
- Pre-run and post-run routines
- Halt and loop capabilities on individual test failure
- Three operating levels

Overview

The LabVIEW Test Executive Toolkit is a multi-purpose LabVIEW add-on package for automated test execution. Using this toolkit, you can control the execution of sequences of tests for production and manufacturing test applications. The Test Executive Toolkit includes a ready-to-run Test Executive application developed in LabVIEW that you can use as delivered or customize to meet your organizations specific requirements and standards. The toolkit delivers the Test Executive in block diagram source code with high-level modules, making it easy to modify to meet your application needs.

Ready-to-Run Test Executive Application

The Test Executive Toolkit also includes a ready-to-run Test Executive application that you can use immediately for your automated test application. This application includes capabilities for test sequencing based upon pass/fail results, logging of test results, ASCII test report generation, three-level password-protected access, and system setup and shutdown routines.

The Test Executive also includes an intuitive, multilevel user interface for displaying test results, operator and UUT serial number prompts, pass/fail banners, and run-time error reporting. A single-window operator interface presents only the information that is relevant to the operator for a given test situation. Three levels of operators, determined by the password string, have access to different levels of functionality. The first level, called Operator, restricts the user to loading sequences from the file and running them to completion. The second level, called Technician, may run individual tests for diagnostic purposes. The third level, called Developer, has access to the full sequence editing capabilities of the Test Executive.

Limit checking and test dependencies give you complete control of your test process. Using an intuitive tabular format, you can specify a set of passes or failures of previous tests required for a given test to run. You can also force the skipping of a test or simulate a pass or fail condition, so you can develop and debug test sequences without being connected to the actual test hardware. You configure all these options using the Sequence and Dependencies Editors included with the Toolkit. Test sequences you create with the Sequence Editor are stored in files for retrieval on the actual test system.

Developed and Delivered in LabVIEW Source Code

The Test Executive is written in the LabVIEW graphical programming language. You receive the block diagrams for block diagrams for all VIs in the Toolkit. Because it is written in the LabVIEW graphical programming language, you can easily modify the Test Executive or any of its components. Common modifications made to the Test Executive include changes to the password criterion, pass/fail banners, UUT serial number prompt, and the test report generation.
The existing program gives you direction on how to implement key Test Executive functions so you dont have to start from scratch. Because the same language is used for both the Test Executive and test programs, you have to learn only one development language to satisfy all your automated test needs.

Part Numbers

LabVIEW Test Executive Toolkit for:

Windows/Windows NT	776731-01
Sun	776731-11
Macintosh	776731-21
HP/UX	776731-31

DatabaseVIEW

Features

- Direct interaction with a local or remote database
- Connection to most popular databases
- High-level, easy-to-use VIs for common database operations
- Complete SQL operation
- Low-level VIs for direct access on columns, records, and tables.

Overview

DatabaseVIEW is a collection of LabVIEW VIs for direct interaction with a local or remote database. High-level Access VIs simplify database access by intelligently encapsulating common database operations into easy-to-use VIs. For example, the Easy SQL VI performs a complete structured query language (SQL) operation on a connected database. First, the SQL statement is executed, then any resulting query data is directly retrieved into the LabVIEW block diagram. Low-level Interface VIs directly access the database, operating on columns and records in database tables.

Structured Query Language

Medium VIs use the SQL, which is a set of commands used to describe, insert, and retrieve information in databases tables. It is a nonprocedural language with which you can refer to and process sets of records in a database. There are three classes of SQL commands statements to define the structure of the database and control access to it; statements to manipulate the content of data tables; and statements used to query or retrieve data from database tables.
Examples of data definition commands include CREATE, ALTER, and DROP, which are used to define, change, and delete tables, respectively. Data manipulation commands include INSERT, UPDATE, and DELETE, which are used to place data into the database record by record, change a particular record, or delete a record. Queries are carried out with the SELECT command. This statement is interpreted as retrieve the columns build_time from the prodstats table, selecting only those rows where the value of the lot column equals the specified value.

Front End and Back End Database Connections

By integrating Medium products into your application productivity will increase for all information intensive problems. You can create front ends to databases that present information with the powerful GUI of LabVIEW. You can create database back ends using LabVIEW data acquisition, instrument drivers, and graphical programming to populate database tables with vital process, test, or research information.

You can more easily create automated test equipment (ATE) systems by using a database to organize and retrieve complex test regiments and also store vast amounts of data and summary results. Manufacturing SPC, and supervisory control and data acquisitions (SCADA) systems benefit from the flexibility of DatabaseVIEW to connect to more than two dozen industry-standard databases, facilitating easy multiapplication integration with a database as an information exchange and repository.

You dont even need to own a database to build these powerful systems. DatabaseVIEW comes with a database engine capable of creating flat-file databases that you can populate with information, update, and query with indexing, record locking, and transaction handling, all directly from LabVIEW. You can even share these database tables among networked LabVIEW applications!

SPC

Features

- Integrate SPC directly into your LabVIEW applications
- Perform online SPC while collecting data or analyze recorded data
- Create control charts
- Display process statistics and histograms
- Perform Pareto analysis

Integrate SPC into Your LabVIEW Data Acquisition Applications

Now you can use LabVIEW not only to monitor your process, but also to identify problems and actually improve the quality of the process. The LabVIEW SPC Toolkit is a VI library for statistical process control (SPC) applications. Using this toolkit, you can apply SPC methods to analyze and track process performance. In addition to subVIs that perform the SPC computations, the toolkit contains numerous example virtual instruments (VIs) and custom controls that demonstrate how to incorporate typical SPC methods and displays into LabVIEW applications.

Capabilities of the SPC Toolkit

The SPC Toolkit includes all the functions you need to incorporate SPC analysis and presentations into your applications. The toolkit addresses three areas of SPC control charts, process statistics, and Pareto analysis. For each of these areas, the toolkit contains analysis functions to compute important information from your data as well as plotting functions for displaying analysis results.

Control Charts

The control chart libraries include VIs that can compute points to be plotted for different types of attributes and variables charts. Select from chart types such as X-bar and s, X-bar and R, p, and np. Apply run rules (AT&T/Western Electric or Nelson) to detect out-of-control points or process shift. The Chart Draw VIs create X-Y graphs with center line and upper and lower limits, control charts with zones, control charts with variable limits, and run and tier plots for displaying the spread of subgroups against specification limits.

Process Statistics

The SPC Toolkit includes several VIs for analyzing the process capability. It features graphing VIs that can fit a normal distribution curve to a histogram to evaluate if the process is normally distributed. Additional VIs calculate process capability ratios, such as Cp and Cpk, as well as fraction nonconforming for normally distributed processes.

Pareto Analysis

You can use Pareto Analysis VIs in the LabVIEW SPC Toolkit to display the relative importance of problems or assignable causes in your process. Create Pareto charts to graph-ically represent the number of occurrences or percentage occurrence of particular causes.

Part Numbers

LabVIEW SPC Toolkit for:

Windows/Windows NT	776954-01
Sun	776954-11
Macintosh	776954-21
HP/UX	776954-31

PID Control

Features

- PI, PD, and PID
- Error-squared PID
- PID with external-reset feedback
- Lead-lag compensation
- Setpoint ramp generation
- Multiloop cascade control
- Feedforward control
- DAQ board example
- Override (minimum/maximum selector) control
- Ratio/bias control

Overview

The PID Control Toolkit adds sophisticated control algorithms to LabVIEW. With this package, you can quickly build data acquisition and control systems. By combining the PID Control Toolkit with the math and logic functions in LabVIEW, you can quickly develop programs for control.

PID Algorithms

The PID VIs implement a wide range of PID algorithms with error-squared and external-reset feedback. VIs also implement lead-lag compensation and setpoint ramp generation. Control parameters include multiloop cascade, feedforward, minimum and maximum override, and ratio/bias. The PID algorithms feature bumpless auto/manual transfer, antireset windup, direct/inverse action, manual output adjustment, and a Run/Hold switch.

Control Strategy Design

Using the PID Control Toolkit, you can design the control strategies of the PID algorithms. You scale I/O values from engineering units to percentages. You can also set up timing of the PID algorithms. Finally, you can use tuning procedures for both Closed-Loop (Ultimate Gain) and Open-Loop (Step Test).

Process Control Examples

The PID Control VIs implement the algorithms using LabVIEW functions and library subVIs without any code interface nodes (CINs), so you can modify the VIs for your applications in LabVIEW without writing any conventional code. This package also includes a complete set of working simulations and demonstrations using National Instruments I/O boards. Documentation for each VI is included in its Get Info... box. In addition, Description boxes document each control and indicator.

Part Numbers

LabVIEW PID Control Toolkit for:

Windows/Windows NT	776634-11
Sun	776634-11
Macintosh	776634-01
HP/UX	776634-31

JTFA

Features

- Award-winning Gabor Spectrogram
- STFT (sliding-window FFT)
- Wigner-Ville Distribution
- Choi-Williams Distribution
- Cone-Shaped Distribution
- Adaptive Spectrogram

Overview

The Joint Time-Frequency Analysis (JTFA) Toolkit is a LabVIEW add-on for precise signal analysis of data whose frequency content changes with time. Applications include speech processing, sound analysis, sonar, radar, vibration analysis, and dynamic signal monitoring. The package also contains the Joint Time-Frequency Analyzer, a run-only application that runs independently from the LabVIEW application, which uses any of the six joint time-frequency algorithms to analyze your stored data files and view the resulting spectrogram on an intensity plot. You may also save the spectrogram to disk for future use.

Joint Time-Frequency Analysis

Traditionally, signals have been analyzed in either the time or the frequency domain, but not jointly in time and frequency. For signals that do not change their spectral content in time, standard spectral analysis reveals what frequencies are present and their relative intensities. For many signals, however, the frequency content changes over time. While Fourier analysis indicates what frequencies were present, it does not show when those frequencies occurred in time. It is the aim of JTFA to describe and determine how the frequencies of nonstationary signals change over time.

Gabor Spectrogram represents the original signal as a linear combination of time-and-frequency-shifted Gaussian functions. After computing the Gabor coefficients, you apply the WVD to the to obtain the spectrogram. The Gabor Spectrogram produces a better SNR in the output intensity plot than the short time Fourier Transform (STFT) because the Gabor coefficients better reflect the local behavior of a signal, resulting in a higher energy concentration in the joint time-frequency domain.

Wigner-Ville Distribution displays the highest resolution of all known JTFA algorithms and is one of the reasons that the Gabor Spectrogram has such good resolution. The Wigner-Ville distribution, however, suffers from interference problems that can be quite severe for some signals.

Choi-Williams Distribution a smoothed version of the Wigner-Ville distribution. This smoothing reduces the interference obtained with the Wigner-Ville distribution at the price of longer computation time.

Cone-Shaped Kernel Distribution considered as a smoothed version of the Wigner-Ville distribution. Although similar to the Choi-Williams distribution, which uses a parabola shape as the smoothing filter, the cone-shaped kernel distribution uses a filter with a cone shape.
Adaptive Spectrogram has the best joint time-frequency resolution for the analysis of quasistationary signals, which are signals that do not change rapidly in time or frequency. The Gabor Spectrogram uses a Gaussian function with uniform variance to compare the analyzed signals at a fixed time-frequency grid, while the adaptive approach adjusts the variance, time, and frequency centers of the Gaussian functions to best match the analysis signal.

Part Numbers

LabVIEW JTFA Toolkit for:

Windows/Windows NT	776733-01
Sun	776733-11
Macintosh	776733-21
HP/UX	776733-31

Picture Control

Features

- Versatile graphics display
- Standard front panel control integration
- Easy-to-use VI library for drawing commands
- High-level examples for common displays

Overview

The LabVIEW Picture Control Toolkit is a versatile graphics add-on package for creating arbitrary front panel displays. The toolkit adds the Picture control and a library of VIs to your LabVIEW system. You can create diagrams using a set of VIs to describe the drawing operations to build these images dynamically. With these tools, you can create new front panel displays like specialized bar graphs, pie charts, and Smith charts. You can also display and even animate arbitrary objects such as robot arms, test equipment, a UUT, or a 2D display of a real-world process.

Examples

The best aspect of the Picture Control Toolkit may be the the examples. They include high-level graph VIs which may be useful as is in your applications. Examples include plotting VI libraries, create waveform, XYand multi-xy displays with scales and grids, and avariety of plot styles (bar, point, line, and so on). There are specialized examples for Polar plots, Smith plots (see Figure 1), and Waterfall plots. In addition, there is a robot arm example that demonstrates animation of real-world processes.
If you build off the examples, youll find many valueable subVIs for drawing waveforms in a variety of styles. You can also create grids and specify range and format for scales. In addition there are also custom controls for most of the example inputs and outputs to aid in creating and passing the picture data.

Part Numbers

LabVIEW Picture Control Toolkit for:

Windows/Windows NT	776732-01
Sun	776732-11
Macintosh	776732-21
HP/UX	776732-31

ASCII Codes

Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL	
01	001	1	SOH	GTL
02	002	2	STX	
03	003	3	ETX	
04	004	4	EOT	SDC
05	005	5	ENQ	PPC
06	006	6	ACK	
07	007	7	BEL	
08	010	8	BS	GET
09	011	9	HT	TCT
0A	012	10	LF	
0B	013	11	VT	
0C	014	12	FF	
0D	015	13	CR	
0E	016	14	SO	
0F	017	15	SI	
10	020	16	DLE	
11	021	17	DC1	LLO
12	022	18	DC2	
13	023	19	DC3	
14	024	20	DC4	DCL
15	025	21	NAK	PPU
16	026	22	SYN	
17	027	23	ETB	
18	030	24	CAN	SPE
19	031	25	EM	SPD
1A	032	26	SUB	
1B	033	27	ESC	
1C	034	28	FS	
1D	035	29	GS	
1E	036	30	RS	
1F	037	31	US	
20	040	32	SP	MLA0
21	041	33	!	MLA1
22	042	34	"	MLA2
23	043	35	#	MLA3
24	044	36	$	MLA4
25	045	37	%	MLA5
26	046	38	&	MLA6
27	047	39	’	MLA7
28	050	40	(	MLA8
29	051	41	)	MLA9
2A	052	42	*	MLA10
2B	053	43	+	MLA11
2C	054	44	,	MLA12
2D	055	45	-	MLA13
2E	056	46	.	MLA14
2F	057	47	/	MLA15
30	060	48	0	MLA16
31	061	49	1	MLA17
32	062	50	2	MLA18
33	063	51	3	MLA19

34	064	52	4	MLA20
35	065	53	5	MLA21
36	066	54	6	MLA22
37	067	55	7	MLA23
38	070	56	8	MLA24
39	071	57	9	MLA25
3A	072	58	:	MLA26
3B	073	59	;	MLA27
3C	074	60	<	MLA28
3D	075	61	=	MLA29
3E	076	62	>	MLA30
3F	077	63	?	UNL
40	100	64	@	MTA0
41	101	65	A	MTA1
42	102	66	B	MTA2
43	103	67	C	MTA3
44	104	68	D	MTA4
45	105	69	E	MTA5
46	106	70	F	MTA6
47	107	71	G	MTA7
48	110	72	H	MTA8
49	111	73	I	MTA9
4A	112	74	J	MTA10
4B	113	75	K	MTA11
4C	114	76	L	MTA12
4D	115	77	M	MTA13
4E	116	78	N	MTA14
4F	117	79	O	MTA15
50	120	80	P	MTA16
51	121	81	Q	MTA17
52	122	82	R	MTA18
53	123	83	S	MTA19
54	124	84	T	MTA20
55	125	85	U	MTA21
56	126	86	V	MTA22
57	127	87	W	MTA23
58	130	88	X	MTA24
59	131	89	Y	MTA25
5A	132	90	Z	MTA26
5B	133	91	[	MTA27
5C	134	92	\	MTA28
5D	135	93	]	MTA29
5E	136	94	^	MTA30
5F	137	95	_	UNT
60	140	96	`	MSA0,PPE
61	141	97	a	MSA1,PPE
62	142	98	b	MSA2,PPE
63	143	99	c	MSA3,PPE
64	144	100	d	MSA4,PPE
65	145	101	e	MSA5,PPE
66	146	102	f	MSA6,PPE
67	147	103	g	MSA7,PPE
68	150	104	h	MSA8,PPE
69	151	105	i	MSA9,PPE
6A	152	106	j	MSA10,PPE
6B	153	107	k	MSA11,PPE

6C	154	108	l	MSA12,PPE
6D	155	109	m	MSA13,PPE
6E	156	110	n	MSA14,PPE
6F	157	111	o	MSA15,PPE
70	160	112	p	MSA16,PPD
71	161	113	q	MSA17,PPD
72	162	114	r	MSA18,PPD
73	163	115	s	MSA19,PPD
74	164	116	t	MSA20,PPD
75	165	117	u	MSA21,PPD
76	166	118	v	MSA22,PPD
77	167	119	w	MSA23,PPD
78	170	120	x	MSA24,PPD
79	171	121	y	MSA25,PPD
7A	172	122	z	MSA26,PPD
7B	173	123	{	MSA27,PPD
7C	174	124	|	MSA28,PPD
7D	175	125	}	MSA29,PPD
7E	176	126	~	MSA30,PPD
7F	177	127	DEL	

How To Contact NI

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a FaxBack system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Note: Before contacting National Instruments for technical support, fill out the technical support form VI.

Bulletin Board Support
FTP Support
FaxBack Support
WWW Support
E-Mail Support (currently U.S. only)
Telephone and Fax Support

Technical Support Form VI

Complete all categories of the Technical Support Form.vi before you contact National Instruments for technical support. Completing this form accurately helps our applications engineers answer your questions more efficiently.

You can advance through the fields of a category by pressing the Tab key.

After completing the form, save it by selecting Save from the File menu, or by pressing the "Save .tsf File" button. The .tsf (technical support form) file, a binary file that can only be opened by the Technical Support Form.vi, contains all of the information that you enter in the Technical Support Form.vi. You must save a .tsf file to retain this information for future problems. The Technical Support Form.vi displays the last .tsf file accessed when it starts.

Update the form each time you make changes to your software or hardware. You can use the same .tsf file for every problem by simply changing the Problem Description, or you can save a different .tsf file for each problem by selecting Save As... from the File menu of the Technical Support Form.vi.

To change the name of a technical support form (.tsf) file, select Save As... from the File menu. If Form name is untitled.tsf, a .tsf file has not been saved to disk. You can save the file by selecting Save from the File menu, or by pressing the "Save .tsf File" button.

After you have completed the technical support form, you can save the information to a text file. To create an ASCII file, select Save Text File... from the File menu, or press the corresponding button. You can then submit this to National Instruments for technical support. For e-mail support, attach the text file you created to your e-mail. For fax support, print the text file you created by using a text editor application, and fax this document. For telephone support, have the completed information available for the support engineer.

You can get help for an item by selecting Show Help from the Help menu of the Technical Support Form.vi, and then placing the cursor over the item.

National Instruments has electronic technical support on the Instrumentation Web, an electronic BBS, and

an FTP site. For more information, select How to Contact NI... from the Help menu of the Technical Support Form.vi.

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422 or (800) 327-3077 Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422 Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 1 48 65 15 59 Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, ftp.natinst.com, as anonymous and use your Internet address, such as joesmith@anywhere.com, as your password. The support files and documents are located in the /support directories.

Instrument drivers are located in the following directories:

LabVIEW for Windows `/support/labview/windows/instruments/`
LabVIEW for Macintosh `/support/labview/mac/instruments/`
LabVIEW for Sun `/support/labview/sun/instruments/`

WWW Support

Connect to `http://www.natinst.com/`

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following numbers:

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

E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com

VXI:	vxi.support@natinst.com
LabVIEW:	lv.support@natinst.com
LabWindows:	lw.support@natinst.com
HiQ:	hiq.support@natinst.com

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.

	Phone	Fax
Australia	03 879 9422	03 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757.00.20	02 757.03.11
Canada-West	519 622-9310	519 622-9311
Canada-East	514 694-8521	514 694-4399
Denmark	45 76 71 11	45 76 26 00
Finland	90 527 2321	90 502 2930
France	1 48 14 24 241	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Italy	02 48301892	02 48301915
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Glossary

A B C D E F G H I K L M N O P Q R S T U V W

Prefix	Meaning	Value
n-	nano-	10^-9
m-	micro-	10^-6
m-	milli-	10^-3
k-	kilo-	10^3
M-	mega-	10^6

Numbers/Symbols

1D One-dimensional.

2D Two-dimensional.

∞ Infinity.

π Pi.

Δ Delta. Difference. Δ*x* denotes the value by which x changes from one index to the next.

A

A
A/D
A16 space
A24 space
A32 space
abort
absolute path
AC
ACFAIL*
active window
ADC
ADC resolution
address
address modifier
address space
address window
AI
AIGND
amplification
AMUX devices
analog input group
analog_io.llb
analog multiplexer
analog output group
analog trigger
analogin
analogout.llb
analysis of variance

ANSI
AO
APDA
application zone
ARMA filter
array
array shell
artificial data dependency
ASCII
ASIC
asserted
ASYNC Protocol
asynchronous
asynchronous execution
ATE
ATN
auto-indexing
autoscaling
autosizing
AZ (application zone)

B

backplane
BCD
BERR*
binary byte stream file
bipolar
bit
bit vector
block diagram
BNF
Boolean controls
BPF
breakpoint
Breakpoint tool
broken VI
BSF
buffer
Bundle node
bus master
bus timeout unit
byte
byte order
byte stream file

C

C string (CStr)
cascading
case
Case Structure
cast
channel
channel clock

chart
CIC
CIN
CIN source code
circular-buffered I/O
client
CLK10
clock
cloning
cluster
cluster shell
code resource
Code Interface Node (CIN)
code width
coercion
coercion dot
Color tool
Color Copy tool
column-major order
command
commands or command messages
Commander
common-mode voltage
communications registers
compile
concatenated Pascal string (CPStr)
conditional retrieval
conditional terminal
configuration registers
connection ID
connector
connector pane
constant
continuous run
control
control flow
controller
Controller or Controller-In-Charge
Controls palette
conversion
conversion device
count terminal
counter.llb
counter/timer group
coupling
CPStr
CPU
CR
CTS
current VI
curve fitting
custom PICT controls and indicators

D

D/A
DAC
daqconf
DARPA
data acquisition
data dependency
data flow
data logging
data or data messages
data space zone
data storage formats
data transfer bus
data type descriptor
datagram
datalog file
DAV
dB
DC
DDE
de-referencing
default handler
default input
default setting
Description box
destination terminal
device
device function
device number
DFT
diagram window
dialog box
DIFF
differential measurement system
digital
digital input group
digital output group
digital trigger
dimension
DIO
DIO1 through DIO8
DIP
DIR
DIRviol
dithering
DLL
DMA
DOR
DORviol
dotted decimal notation
down counter
drag
DRAM
driver
DS (data space) zone
DSP
DSR

DTR
DUT

E

EABO
ECIC
ECL
EEPROM
EISA
embedded controller
empty array
END
EOF
EOI
EOL
EOS
ESAC
ETAB
ethernet
event
event signal
executable
execution highlighting
Extended Class device
extended controller
Extended Longword Serial Protocol
external controller
external routine
external trigger

F

FFT
FHS
FHT
FIFO
file refnum
filtering
FIR filter
flattened data
floating signal sources
flow-through parameters
For Loop
Formula Node
FPU
frame
free label
front panel
front panel window
function
Functions palette

G

G
gain
GATE input pin
General Purpose Interface Bus
global variable
glyph
GMT
GPIB
GPIB address
GPIB board
GPIO
graph control and indicator
grounded measurement system
grounded signal sources
group
GTL

H

handle
handler
handshake
handshaked digital I/O
handshaking
hardware context
hardware triggering
Help window
hex
hierarchical menu
hierarchical palette
Hierarchy window
high-level
high-level function
histogram
housing
HPF
Hz

I

I/O
IAC
IACK
icon
Icon Editor
icon pane
IDY
IEEE
IEEE 1014
IIR filter
immediate digital I/O
indicator
Inf
inplace
input limits

input range
instrument driver
interface message
internet
internetwork
interrupt
interrupt handler
interrupter
interval scanning
IP
ISA
ISO
isolation
ist
iteration terminal

K

Kwords

L

label
Labeling tool
LabVIEW
LabVIEW string (LStr)
latched digital I/O
LED
legend
LF
limit settings
linearization
Listener
LLO
logical address
LOK
longword
Longword Serial Protocol
low-level
low-level function
LPF
LSB

M

m
MA
mapping
marquee
master
matrix
Maxint
MB
mechanical-action controls and indicators
median filter

memory buffer
Memory Class device
menu bar
Message-Based device
meta-click
MLA
mnemonic
MODID
modular programming
MPW
MQE
MSA
MSB
MSE
MTA
multibuffered I/O
multiplexed mode
multiplexer
multitasking

N

NaN
NB
NDAC
NI-DAQ
NI-PNP.EXE
NI-PNP.INI
NI-VXI
nodes
non-referenced signal sources
Non-referenced single-ended (NRSE) measurement system
nondisplayable characters
nondisplayable indicators
nonlatched digital I/O
nonprivileged access
nonsingular matrix
not-a-path
not-a-refnum
NRFD
NRSE
numeric controls and indicators
Nyquist frequency

O

object
object code
Object pop-up menu tool
oct
OLE
OLE Automation
onboard channels
Operating tool
OUT output pin

output limits

P

palette
palette menu
panel window
parallel mode
parallel poll configure
parallel poll disable
parallel poll enable
parallel poll unconfigure
Pascal string (PStr)
path
pattern generation
PC
peek
PGIA
pixmap
platform
plot
Plug and Play devices
pointer
poke
polymorphism
pop up
pop-up menus
portable
Positioning tool
postriggering
PPC
PPD
PPE
PPU
pretriggering
private data structures
privileged access
probe
Probe tool
programmatic printing
protocol
pseudocode
pull-down menus
pulse trains
pulsed output

Q

query
queue

R

RAM
read

read mark
read mode
recursive filter
reentrant execution
reference
referenced signal sources
referenced single-ended (RSE) measurement system
refnum
register
Register-Based device
regression analysis
relocatable
remote address
REN
representation
REQF
REQT
resizing handles
Resource Manager
response signal
ring control
ripple
RM
RMS
ROAK
ROR
RORA
row-major order
RR
RRviol
RSE
RTD
RTS
RTSI
run_me.llb

S

sample
sample counter
scalar
scale
scan
scan clock
scan rate
scan width
scope chart
SCPI
Scroll tool
SCSI
SCXI
sec
SEMI-SYNC Protocol
sequence local
Sequence Structure

serial poll
Servant
server
settling time
shared external subroutine
Shared Memory Protocol
shift register
short integer
signal
signal conditioning
signal divider
signed integer
simple-buffered I/O
single-ended inputs
sink terminal
slave
slider
SMP
software trigger
software triggering
source code
SOURCE input pin
source terminal
spreadsheet
SRQ
status byte
status/ID
STC
strain gauge
string
string controls and indicators
strip chart
structure
STST
stub VI
subdiagram
subVI
supervisory access
SVD
sweep chart
switchless device
SYNC Protocol
synchronous communications
syntax
SYSFAIL*
SYSRESET*
System Controller
system hierarchy

T

table-driven execution
Talker
task
task ID

TC
TCP
TCP/IP
terminal
TIC
tick
timeout
toggled output
tool
toolbar
Tools palette
top-level VI
track-and-hold
transducer excitation
trigger
tristated
TTL
tunnel
type descriptor

U

UART
UDP
unasserted
unipolar
universal constant
UNL
unsigned integer
UnSupCom
UNT
update
update rate
update width
user-defined constant
UT
utility
UUT

V

V
VDC
vector
VI
VIC
virtual instrument (VI)
VISA
VME
VMEbus Class device
Vref
VXIbus
vxiedit

W

waveform
WDAQCONF.EXE
While Loop
windowing FIR filter design
wire
Wiring tool
word
Word Serial Protocol
WR
write
write mark
WRviol

A16 space

One of the VXIbus address spaces. Equivalent to the VME 64 KB short address space. In VXI, the upper 16 KB of A16 space is allocated for use by the configuration registers of the VXI device. This 16 KB region is referred to as VXI configuration space.

A24 space

One of the VXIbus address spaces. Equivalent to the VME 16 MB standard address space.

A32 space

One of the VXIbus address spaces. Equivalent to the VME 4 GB extended address space.

A

amperes.

abort

The procedure that terminates a program when a mistake, malfunction, or error occurs.

absolute path

Relative file or directory path that describes the location relative to the top of level of the file system.

AC

Alternating current.

ACFAIL*

A VMEbus backplane signal that is asserted when a power failure has occurred (either AC line source or power supply malfunction), or if it is necessary to disable the power supply (such as for a high temperature condition).

active window

Window that is currently set to accept user input. Usually the front window. The title bar of an active window is highlighted. You make a window active by clicking on it, or be selecting it from the **Windows** menu.

A/D

Analog-to-digital.

ADC

Analog-to-digital converter. An electronic device, often an integrated circuit, that converts an analog voltage to a digital number.

ADC resolution

The resolution of the ADC, which is measured in bits. An ADC with 16 bits has a higher resolution, and thus a higher degree of accuracy than a 12-bit ADC.

address

Character code that identifies a specific location (or series of locations) in memory.

address modifier

One of six signals in the VMEbus specification used by VMEbus masters to indicate the address space and mode (supervisory/nonprivileged, data/program/block) in which a data transfer is to take place.

address space

A set of 2n memory locations differentiated from other such sets in VXI/VMEbus systems by six signal lines known as address modifiers. *n* is the number of address lines required to uniquely specify a byte location in a given space. Valid numbers for *n* are 16, 24, and 32.

address window

A range of address space that can be accessed from the application program.

AI

Analog input.

AIGND

The analog input ground pin on a DAQ device.

amplification

A type of signal conditioning that improves accuracy in the resulting digitized signal and to reduce noise.

AMUX devices

See analog multiplexers.

analogin

A LabVIEW DAQ library containing VIs that perform analog input with DAQ devices and SCXI modules and can write or stream the acquired data to disk.

analog_io.llb

A LabVIEW DAQ library containing VIs for analog I/O control loops.

analog input group

A collection of analog input channels. You can associate each group with its own clock rates, trigger and buffer configurations, and so on. A channel cannot belong to more than one group.

Because each board has one ADC, only one group can be active at any given time. That is, once a control VI starts a timed acquisition with group *n*, subsequent control and read calls must also refer to group *n*. You use the task ID to refer to the group.

analog multiplexer

Devices that increase the number of measurement channels while still using an single instrumentation amplifier. Also called AMUX devices.

analogout.llb

A LabVIEW DAQ library containing VIs that generate single values or multiple values (waveforms) to output through analog channels.

analog output group

A collection of analog output channels. You can associate each group with its own clock rates, buffer configurations, and so on. A channel cannot belong to more than one group.

analog trigger

A trigger that occurs at a user-selected point on an incoming analog signal. Triggering can be set to occur at a specified level on either an increasing or a decreasing signal (positive or negative slope).

analysis of variance

Analysis of whether the level of the factor has an effect on the outcome of an experiment.

ANSI

American National Standards Institute.

AO

Analog output.

APDA

Apple Programmer Developer Association.

application zone

See AZ .

ARMA filter

Autoregressive moving-average filter. *See* IIR filter.

array

Ordered, indexed set of data elements of the same type.

array shell

Front panel object that houses an array. It consists of an index display, a data object window, and an optional label. It can accept various data types.

artificial data dependency

Condition in a dataflow programming language in which the arrival of data rather than its value triggers execution of a node. *See also* data dependency.

ASCII

American Standard Code for Information Interchange.

ASIC

Application Specific Integrated Circuit (a custom chip)

asserted

A signal in its active true state.

ASYNC Protocol

A two-device, two-line handshake trigger protocol using two consecutive even/odd trigger lines (a source/acceptor line and an acknowledge line).

asynchronous

Not synchronized; not controlled by periodic time signals, and therefore unpredictable with regard to the timing of execution of commands.

asynchronous execution

Mode in which multiple processes share processor time. For example, one process executes while the others wait for interrupts, as while performing device I/O or 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 determines maximum and minimum scale values, as well.

autosizing

Automatic resizing of labels to accommodate text that you enter.

AZ (application zone)

Memory allocation section that holds all data in a VI except execution data.

backplane

An assembly, typically a printed circuit board, with 96-pin connectors and signal paths that bus the connector pins. A C-size VXIbus system will have two sets of bused connectors called the J1 and J2 backplanes. A D-size VXIbus system will have three sets of bused connectors called the J1, J2, and J3 backplane.

BCD

Binary-coded decimal.

BERR*

Bus Error signal. This signal is asserted by either a slave device or the BTO unit (bus timeout unit) when an incorrect transfer is made on the Data Transfer Bus (DTB). The BERR* signal is also used in VXI for certain protocol implementations such as writes to a full Signal register and synchronization under the Fast Handshake Word Serial Protocol.

binary byte stream file

A byte stream file that LabVIEW writes as a sequence of typed data (such as 32-bit integers) instead of characters. *See* byte stream file.

bipolar

A signal range that includes both positive and negative values (for example, -5 to 5 V).

bit

Binary digit. The smallest possible unit of data: a two-state, yes/no, 0/1 alternative. The building block of binary coding and numbering systems. Several bits make up a *byte*.

bit vector

A string of related bits in which each bit has a specific meaning.

block diagram

Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the virtual instrument. The block diagram resides in the block diagram of the VI.

BNF

Backus-Naur Form. A common representation for language grammars in computer science.

Boolean controls

Front panel objects used to manipulate and display or input and and indicators output Boolean (True or False) data. Several styles are available, such as switches, buttons and LEDs.

BPF

Bandpass filter.

breakpoint

Mode that halts execution when a subVI is called. You set a breakpoint by clicking on the toolbar and then on a node.

Breakpoint tool

Tool used to set a breakpoint on a VI, node, or wire.

broken VI

VI that cannot be compiled or run; signified by a run button with a broken arrow.

BSF

Bandstop filter.

buffer

Temporary storage for acquired or generated data.

Bundle node

Function that creates clusters from various types of elements.

bus master

A device that is capable of requesting the Data Transfer Bus (DTB) for the purpose of accessing a slave device.

bus timeout unit

A VMEbus functional module that times the duration of each data transfer on the Data Transfer Bus (DTB) and terminates the DTB cycle if the duration is excessive. Without the termination capability of this module, a bus master attempt to access a nonexistent slave could result in an indefinitely long wait for a slave response.

byte

A grouping of adjacent binary digits operated on by the computer as a single unit. In VXI systems, a byte consists of 8 bits.

byte order

How bytes are arranged within a word or how words are arranged within a longword. Motorola ordering stores the most significant byte (MSB) or word first, followed by the least significant byte (LSB) or word. Intel ordering stores the LSB or word first, followed by the MSB or word.

byte stream file

A file that stores data as a sequence of ascii characters, or bytes.

C string (CStr)

A series of zero or more unsigned characters, terminated by a zero, used in the C programming language.

cascading

Process of extending the counting range of a counter chip by connecting to the next higher counter.

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.

channel

Pin or wire lead to which you apply or from which you read the analog or digital signal. Analog signals can be single-ended or differential. For digital signals, you group channels to form ports. Ports usually consist of either four or eight digital channels.

channel clock

The clock controlling the time interval between individual channel sampling within a scan. Boards with simultaneous sampling do not have this clock.

chart

See scope chart, strip chart, and sweep chart.

CIN

See Code Interface Node.

CIN source code

Original, uncompiled text code. *See* object code.

circular-buffered I/O

Input/output operation that reads or writes more data points than can fit in the buffer. When LabVIEW reaches the end of the buffer, LabVIEW returns to the beginning of the buffer and continues to transfer data.

client

The application that sends or calls messages from the server application in a dynamic data exchange.

CLK10

A 10-MHz, ± 100-ppm, individually buffered (to each module slot), differential ECL system clock that is sourced from Slot 0 and distributed to Slots 1 through 12 on P2. It is distributed to each slot as a single-source, single-destination signal with a matched delay of under 8 nsec.

clock

Hardware component that controls timing for reading from or writing to groups.

cloning

To make a copy of a control or some other LabVIEW object by <*Key*>-clicking on it and dragging the copy to its new location. In Windows, click on the object with the left mouse button while holding down the <Ctrl> key and drag the copy to its new location. On the Macintosh, <option>-click on the object and drag the copy to its new location. On the Sun, click the left mouse button while holding down the <meta> key, and drag the copy to its new location, or click on the object with the middle mouse button and drag the copy.

cluster

A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.

cluster shell

Front panel object that contains the elements of a cluster.

Code Interface Node (CIN)

Special block diagram node through which you can link conventional, text-based code to a VI.

code resource

Resource that contains executable machine code. You link code resources to LabVIEW through a CIN.

code width

The smallest detectable change in an input voltage of a DAQ device.

coercion

The automatic conversion LabVIEW performs to change the numeric representation of a data element.

coercion dot

Glyph on a node or terminal indicating that the numeric representation of the data element changes at that point.

Color tool

Tool you use to color objects and backgrounds.

Color Copy tool

Copies colors for pasting with the Color tool.

column-major order

A way to organize the data in a 2D array by columns.

command

A directive to a device. In VXI, three types of commands are as follows:
In Word Serial Protocol, a 16-bit imperative to a Servant from its Commander (written to the Data Low register);
In Shared Memory Protocol, a 16-bit imperative from a client to a server, or vice versa (written to the Signal register);
In Instrument devices, an ASCII-coded, multi-byte directive.

Commander

A Message-Based device which is also a bus master and can control one or more Servants.

common-mode voltage

Any voltage present at the instrumentation amplifier inputs with respect to amplifier ground.

communications registers

In Message-Based devices, a set of registers that are accessible to the device's Commander and are used for performing Word Serial Protocol communications.

compile

Process that converts high-level code to machine-executable code. LabVIEW automatically compiles VIs before they run for the first time after creation or alteration.

concatenated Pascal string (CPStr)

A list of Pascal-type strings concatenated into a single block of memory.

conditional retrieval

A method of triggering in which you to simulate an analog trigger using software. Also called software triggering.

conditional terminal

The terminal of a While Loop that contains a Boolean value that determines whether the VI performs another iteration.

configuration registers

A set of registers through which the system can identify a module device type, model, manufacturer, address space, and memory requirements. In order to support automatic system and memory configuration, the VXIbus specification requires that all VXIbus devices have a set of such registers.

connection ID

A unique identification of a connection that you use for reference in subsequent VI calls.

connector

Part of the VI or function node that contains its input and output terminals, through which data passes to and from the node.

connector pane

Region in the upper right corner of a front panel window that displays the VI's connector. It underlies the Icon pane.

constant

See universal and user-defined constants.

continuous run

Execution mode in which a VI is run repeatedly until the operator stops it. You enable it by clicking on the continuous run button.

control

Front panel object for entering data to a VI interactively or to a subVI programmatically.

control flow

Programming system in which the sequential order of instructions determines execution order. Most conventional text-based programming languages, such as C, Pascal, and BASIC, are control flow languages.

controller

An intelligent device (usually involving a CPU) that is capable of controlling other devices.

Controls palette

Menu of controls and indicators.

conversion

Changing the type of a data element.

conversion device

Device that transforms a signal from one form to another. For example, analog-to-digital converters (ADCs) for analog input, digital-to-analog converters (DACs) for analog output, digital input or output ports, and counter/timers are conversion devices.

counter.llb

A LabVIEW DAQ library containing VIs that count the rising and falling edges of TTL signals, generate TTL pulses, and measure the frequency and period of TTL signals.

count terminal

The terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.

counter/timer group

A collection of counter/timer channels. You can use this type of group for simultaneous operation of multiple counter/timers.

coupling

The manner in which a signal is connected from one location to another.

CPStr

See concatenated Pascal string.

CPU

Central processing unit.

CR

Carriage Return; the ASCII character 0Dh.

current VI

VI whose front panel, block diagram, or Icon Editor is the active window.

curve fitting

Technique for extracting a set of curve parameters or coefficients from a data set to obtain a functional description of the data set.

custom PICT controls and indicators

Controls and indicators whose parts can be replaced by graphics you supply.

D/A

Digital-to-analog.

DAC

Digital-to-analog converter. An electronic device, often an integrated circuit, that converts a digital number into a corresponding analog voltage or current.

daqconf

The NI-DAQ configuration utility on the Sun.

DARPA

Defense Advanced Research Projects Agency.

data acquisition

Process of acquiring data, typically from A/D or digital input plug-in boards.

data dependency

Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *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. LabVIEW is a dataflow system.

data logging

Generally, to acquire data and simultaneously store it in a disk file. LabVIEW file I/O functions can also log data.

data space zone

See DS zone.

data storage formats

The arrangement and representation of data stored in memory.

data transfer bus

One of four buses on the VMEbus backplane. The DTB is used by a bus master to transfer binary data between itself and a slave device.

data type descriptor

Code that identifies data types, used in data storage and representation.

datagram

IP-packaged data components that contain, among other things, the data and a header that indicates the source and destination addresses.

datalog file

A 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.

dB

Decibels.

DC

Direct current.

DDE

Dynamic Data Exchange. A client-controlled Windows protocol for communication between applications.

default handler

Automatically installed at start-up to handle associated interrupt conditions; the software can then replace it with a specified handler.

default input

The default value of a front panel control.

default setting

A default parameter value recorded in the driver. In many cases, the default input of a control is a certain value (often 0) that means *use the current default setting*. For example, the default input for a parameter may be *do not change current setting*, and the default setting may be *no AMUX-64T boards*. If you do change the value of such a parameter, the new value becomes the new setting. You can set default settings for some parameters in the configuration utility.

de-referencing

Accessing the contents of the address location pointed to by a pointer.

Description box

Online documentation for a LabVIEW object.

destination terminal

See sink terminal.

device

A DAQ device inside your computer or attached directly to your computer through a parallel port. Plug-in boards, PCMCIA cards, and devices such as the DAQPad-1200, which connects to your computer's parallel port, are all examples of DAQ devices. SCXI modules are distinct from devices, with the exception of the SCXI-1200, which is a hybrid.

device number

The slot number or board ID number assigned to the board when you configured it. \

DFT

Discrete Fourier transform.

diagram window

VI window that contains the VI's block diagram code.

dialog box

An interactive screen with prompts in which the user specifies additional information needed to complete a command.

DIFF

Differential. A differential input is an analog input consisting of two terminals, both of which are isolated from computer ground and whose difference you measure.

differential measurement system

A way you can configure your device to read signals, in which you do not need to connect either input to a fixed reference, such as the earth or a building ground.

digital

A LabVIEW DAQ library containing VIs that perform immediate digital I/O and digital handshaking with DAQ devices and SCXI modules .

digital input group

A collection of digital input ports. You can associate each group with its own clock rates, handshaking modes, buffer configurations, and so on. A port cannot belong to more than one group.

digital output group

A collection of digital output ports. You can associate each group with its own clock rates, handshaking modes, buffer configurations, and so forth. A port cannot belong to more than one group.

digital trigger

A TTL level signal having two discrete levels--a high and a low level.

dimension

Size and structure attribute of an array.

DIP

Dual inline package.

DIR

Data In Ready

DIRviol

Data In Ready violation

dithering

The addition of Gaussian noise to an analog input signal.

DLL

Dynamic link library.

DMA

Direct memory access. A method by which data you can transfer data to computer memory from a device or memory on the bus (or from computer memory to a device) while the processor does something else. DMA is the fastest method of transferring data to or from computer memory.

DOR

Data Out Ready

DORviol

Data Out Ready violation

dotted decimal notation

A method of describing a 32-bit internet address in which the address is divided into four 8-bit binary numbers and written as four integers separated by decimal points.

down counter

Performing frequency division on an internal signal.

drag

To drag the mouse cursor on the screen to select, move, copy, or delete objects.

driver

Software that controls a specific hardware device, such as a data acquisition board.

DRAM

Dynamic RAM (Random Access Memory); storage that the computer must refresh at frequent intervals.

DS (data space) zone

Memory allocation section that holds VI execution data.

DSP

Digital signal processing.

DUT

Device under test.

ECL

Emitter-Coupled Logic

EEPROM

Electrically erased programmable read-only memory. Read-only memory that you can erase with an electrical signal and reprogram.

EISA

Extended Industry Standard Architecture.

embedded controller

An intelligent CPU (controller) interface plugged directly into the VXI backplane, giving it direct access to the VXIbus. It must have all of its required VXI interface capabilities built in.

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

Signals the end of a data string.

EOF

End-of-file. Character offset of the end of file relative to the beginning of the file (that is, the EOF is the size of the file).

EOL

End of line character.

EOS

End Of String; a character sent to designate the last byte of a data message.

ethernet

A network system that carries audio and video information as well as computer data.

event

The condition or state of an analog or digital signal.

event signal

A 16-bit value written to a Message-Based device's Signal register in which the most significant bit (bit 15) is a 1, designating an Event (as opposed to a Response signal). The VXI specification reserves half of the Event values for definition by the VXI Consortium. The other half are user defined.

executable

A stand-alone piece of code that will run, or execute.

execution highlighting

Feature that animates VI execution to illustrate the data flow in the VI.

Extended Class device

A class of VXIbus device defined for future expansion of the VXIbus specification. These devices have a subclass register within their configuration space that defines the type of extended device.

extended controller

A mainframe extender with additional VXIbus controller capabilities.

Extended Longword Serial Protocol

A form of Word Serial communication in which Commanders and Servants communicate with 48-bit data transfers.

external controller

In this configuration, a plug-in interface board in a computer is connected to the VXI mainframe via one or more VXIbus extended controllers. The computer then exerts overall control over VXIbus system operations.

external routine

See shared external routine.

external trigger

A voltage pulse from an external source that triggers an event such as A/D conversion.

FFT

Fast Fourier transform.

FHT

Fast Hartley transform.

FHS

Fast Handshake; a mode of the Word Serial Protocol which uses the VXIbus signals DTACK* and BERR* for synchronization instead of the Response register bits.

FIFO

First In-First Out; a method of data storage in which the first element stored is the first one retrieved.

file refnum

An identifier that LabVIEW associates with a file when you open it. You use the file refnum to specify that you want a function or VI to perform an operation on the open file.

filtering

A type of signal conditioning that allows you to filter unwanted signals from the signal you are trying to measure.

FIR filter

Finite impulse response filter. A digital filter whose impulse response is finite. FIR filters are also known as nonrecursive filters, convolution filters, or moving-average (MA) filters.

flattened data

Data of any type that has been converted to a string, usually for writing it to a file.

floating signal sources

Signal sources with voltage signals that are not connected to an absolute reference or system ground. Also called nonreferenced signal sources. Some common example of floating signal sources are batteries, transformers, or thermocouples.

flow-through parameters

Parameters that return the same value as an input parameter.

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 É .

Formula Node

Node that executes formulas that you enter as text. This node is especially useful for lengthy formulas that would be cumbersome to build in block diagram form.

FPU

Floating-point coprocessor.

frame

Subdiagram of a Sequence Structure.

free label

Label on the front panel or block diagram that does not belong to any other object.

front panel

The interactive user interface of a VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.

front panel window

VI window that contains the front panel, the execution palette and the icon/connector pane.

function

Built-in execution element, comparable to an operator, function, or statement in a conventional language.

Functions palette

Palette containing block diagram structures, constants, controls, communication features, and VIs.

G

LabVIEW graphical programming language.

gain

The amplification or attenuation of a signal.

GATE input pin

A counter input pin that controls when counting in your application occurs.

global variable

Non-reentrant subVI with local memory that uses an uninitialized shift register to store data from one execution to the next. The memory of copies of these subVIs is shared and thus can be used to pass global data between them.

glyph

A small picture or icon.

GMT

Greenwich Mean Time.

GPIB

General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.

GPIO

General Purpose Input Output, a module within the National Instruments TIC chip which is used for two purposes. First, GPIOs are used for connecting external signals to the TIC chip for routing/conditioning to the VXIbus trigger lines. Second, GPIOs are used as part of a crosspoint switch matrix.

graph control and indicator

Front panel object that displays data in a Cartesian plane.

grounded measurement system

See referenced single-ended measurement system.

grounded signal sources

Signal sources with voltage signals that are referenced to a system ground, such as the earth or a building ground. Also called referenced signal sources

group

A collection of input or output channels or ports that you define. Groups can contain analog input, analog output, digital input, digital output, or counter/timer channels. A group can contain only one type of channel, however. You use a task ID number to refer to a group after you create it. You can define up to 16 groups at one time.

To erase a group, you pass an empty channel array and the group number to the group configuration VI. You do not need to erase a group to change its membership. If you reconfigure a group whose task is active, LabVIEW clears the task and returns a warning. LabVIEW does not restart the task after you reconfigure the group.

handle

Pointer to a pointer to a block of memory; handles reference arrays and strings. An array

handler

A device driver installed as part of the operating system of the computer.

handshaked digital I/O

A type of digital acquisition/generation where a device or module accepts or transfers data after a digital pulse has been received. Also called latched digital I/O.

handshaking

A type of protocol that makes it possible for two devices to synchronize operations.

hardware context

The hardware setting for address space, access privilege, and byte ordering.

hardware triggering

A form of triggering where you set the start time of an acquisition and gather data at a known position in time relative to a trigger signal.

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.

high-level

Programming with instructions in a notation more familiar to the user than machine code. Each high-level statement corresponds to several low-level machine code instructions and is machine independent, meaning that it is portable across many platforms.

hierarchical palette

Palette that contains palettes and subpalettes.

hierarchical menu

Menu that contains submenus or palettes.

Hierarchy window

Windows that graphically displays the hierarchy of VIs and subVIs.

histogram

Frequency count of the number of times that a specified interval occurs in the input sequence.

housing

Nonmoving part of front panel controls and indicators that contains sliders and scales.

HPF

Highpass filter.

Hz

Hertz. The number of scans read or updates written per second.

IAC

Interapplication Communication. A feature of Apple Macintosh system software version 7 by which applications can communicate with each other.

IACK

Interrupt Acknowledge

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-hand corner of the front panel and block diagram windows that displays the VI icon.

IEEE

Institute of Electrical and Electronic Engineers.

IEEE 1014

The VME specification.

IIR filter

Infinite impulse response filter. An IIR filter is a digital filter whose impulse response can theoretically be infinite in duration. IIR filters are also known as recursive filters or autoregressive moving-average (ARMA) filters.

immediate digital I/O

A type of digital acquisition/generation where LabVIEW updates the digital lines or port states immediately or returns the digital value of an input line. Also called nonlatched digital I/O.

indicator

Front panel object that displays output.

Inf

Digital display value for a floating point representation of infinity.

inplace

Characteristic of an operation whose input and output data can use the same memory space.

input limits

The upper and lower voltage inputs for a channel. You must use a pair of numbers to express the input limits. The VIs can infer the input limits from the input range, input polarity, and input gain(s). Similarly, if you wire the input limits, range, and polarity, the VIs can infer the onboard gains when you do not use SCXI.

input range

The difference between the maximum and minimum voltages an analog input channel can measure at a gain of 1. The input range is a scalar value, not a pair of numbers. By itself the input range does not uniquely determine the upper and lower voltage limits. An input range of 10ÊV could mean an upper limit of +10 V and a lower of 0 V or an upper limit of +5 V and a lower limit of -5 V.

The combination of input range, polarity, and gain determines the input limits of an analog input channel. For some boards, jumpers set the input range and polarity, while you can program them for other boards. Most boards have programmable gains. When you use SCXI modules, you also need their gains to determine the input limits.

instrument driver

VI that controls a programmable instrument.

internet

See internetwork.

internetwork

Single or interconnected networks.

interrupt

A signal indicating that the central processing unit should suspend its current task to service a designated activity.

interrupt handler

A functional module that detects interrupt requests generated by interrupters and performs appropriate actions.

interrupter

A device capable of asserting interrupts and responding to an interrupt acknowledge cycle.

interval scanning

Scanning method where there is a longer interval between scans than there is between individual channels comprising a scan.

I/O

Input/output. The transfer of data to or from a computer system involving communications channels, operator interface devices, and/or data acquisition and control interfaces.

IP

Internet Protocol. Protocol that performs the low-level service of packaging data into components (datagrams). *See* TCP/IP.

ISA

Industry Standard Architecture.

isolation

A type of signal conditioning in which you isolate the transducer signals from the computer for safety purposes. This protects you and your computer from large voltage spikes and makes sure the measurements from the DAQ device are not affected by differences in ground potentials.

iteration terminal

The terminal of a For Loop or While Loop that contains the current number of completed iterations.

Kwords

1,024 words of memory.

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.

LabVIEW string (LStr)

The string data type used by LabVIEW block diagrams.

latched digital I/O

A type of digital acquisition/generation where a device or module accepts or transfers data after a digital pulse has been received. Also called handshaked digital I/O.

LED

Light-emitting diode.

legend

Object owned by a chart or graph that display the names and plot styles of plots on that chart or graph.

LF

Line feed.

limit settings

The maximum and minimum voltages of the analog signals you are measuring or generating.

linearization

A type of signal conditioning in which LabVIEW linearizes the voltage levels from transducers, so the voltages can be scaled to measure physical phenomena.

logical address

An 8-bit number that uniquely identifies the location of each VXIbus device's configuration registers in a system. The A16 register address of a device is C000h + Logical Address * 40h.

longword

Data type of 32-bit integers.

Longword Serial Protocol

A form of Word Serial communication in which Commanders and Servants communicate with 32-bit data transfers instead of 16-bit data transfers as in the normal Word Serial Protocol.

low-level

Programming at the system level with machine-dependent commands.

LPF

Lowpass filter.

LSB

Least significant bit.

MA

Moving average. *See* FIR filter.

mapping

Establishing a range of address space for a one-to-one correspondence between each address in the window and an address in VXIbus memory.

marquee

A moving, dashed border that surrounds selected objects.

master

A functional part of a MXI/VME/VXIbus device that initiates data transfers on the backplane. A transfer can be either a read or a write.

matrix

Two-dimensional array.

Maxint

The maximum value that an integer, data type can represent.

MB

Megabytes of memory.

mechanical-action controls and indicators

Front panel objects that look and operate like familiarmechanical or electro-mechanical devices. Examples include toggle switches, slides, meters, knobs, and LEDs

median filter

A nonlinear filter that removes high frequency noise while preserving edge information.

memory buffer

See buffer.

Memory Class device

A VXIbus device that, in addition to configuration registers, has memory in VME A24 or A32 space that is accessible through addresses on the VME/VXI data transfer bus.

Message-Based device

An intelligent device that implements the defined VXIbus registers and communication protocols. These devices are able to use Word Serial Protocol to communicate with one another through communication registers.

meta-click

On the Sun, to click the mouse button while pressing the <meta> key.

menu bar

Horizontal bar that contains names of main menus.

modular programming

Programming that uses interchangeable computer routines.

mnemonic

A string associated with an integer value.

MODID

A set of 13 signal lines on the VXI backplane that VXI systems use to identify which modules are located in which slots in the mainframe.

MPW

Macintosh ProgrammerÕs Workshop.

MQE

Multiple Query Error; a type of Word Serial Protocol error. If a Commander sends two Word Serial queries to a Servant without reading the response to the first query before sending the second query, a MQE is generated.

MSB

Most significant bit.

MSE

Mean squared error. The MSE is a relative measure of the residuals between the expected curve values and the actual observed values.

multibuffered I/O

Input operation for which you allocate more than one memory buffer so you can read and process data from one buffer while the acquisition fills another.

multiplexed mode

An SCXI operating mode in which analog input channels are multiplexed into one module output so that your cabled DAQ device has access to the moduleÕs multiplexed output as well as the outputs on all other multiplexed modules in the chassis through the SCXI bus. Also called serial mode.

multiplexer

A set of semiconductor or electromechanical switches with a common output that can select one of a number of input signals and that you commonly use to increase the number of signals measured by one ADC.

multitasking

The ability of a computer to perform two or more functions simultaneously without interference from one another. In operating system terms, it is the ability of the operating system to execute multiple applications/processes by time-sharing the available CPU resources.

NaN

Digital display value for a floating-point representation of not a number, typically the result of an undefined operation, such as log(-1).

NB

NuBus.

NI-DAQ

The NI-DAQ configuration utility on the Macintosh.

NI-PNP.EXE

A stand-alone executable that NI-DAQ installs in your NI-DAQ root drive that detects and configures any Plug and Play devices you have in your computer.

NI-PNP.INI

A file, generated by the NI-PNP. EXE, that contains information about all the National Instruments devices in your computer, including Plug and Play devices.

NI-VXI

The National Instruments bus interface software for VME/VXIbus systems.

nodes

Execution elements of a block diagram consisting of functions, structures, and subVIs.

nondisplayable characters

ASCII characters that you cannot display, such as newline, tab, and so on.

nondisplayable indicators

ASCII characters that cannot be displayed, such as ESC, NUL, SOH, indicators and so on.

nonlatched digital I/O

A type of digital acquisition/generation where LabVIEW updates the digital lines or port states immediately or returns the digital value of an input line. Also called immediate digital I/O.

nonprivileged access

One of the defined types of VMEbus data transfers; indicated by certain address modifier codes. Each of the defined VMEbus address spaces has a defined nonprivileged access mode.

non-referenced signal sources

Signal sources with voltage signals that are not connected to an absolute reference or system ground. Also called floating signal sources. Some common example of non-referenced signal sources are batteries, transformers, or thermocouples.

nonsingular matrix

Matrix in which no row or column is a linear combination of any other row or column, respectively.

NRSE

Nonreferenced single-ended.

Non-referenced single-ended (NRSE) measurement system

All measurements are made with respect to a common reference, but the voltage at this reference can vary with respect to the measurement system ground.

not-a-path

A predefined value for the path control that means the path is invalid.

not-a-refnum

A predefined value that means the refnum is invalid.

numeric controls and indicators

Front panel objects used to manipulate and display or input and output numeric data.

Nyquist frequency

The highest frequency a DSP system can process.

object

Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures.

object code

Compiled version of source code. Object code is not stand-alone because you must load it into LabVIEW to run it.

Object pop-up menu tool

Tool used to access an objectÕs pop-up menu.

OLE

Object Linking and Embedding.

OLE Automation

A feature which allows LabVIEW to access objects by automation servers in the system.

onboard channels

Channels provided by the plug-in data acquisition board.

Operating tool

Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

OUT output pin

A counter output pin where the counter can generate various TTL pulse waveforms.

output limits

The upper and lower voltage or current outputs for an analog output channel. The output limits determine the polarity and voltage reference settings for a board.

palette

Menu that displays a palette of pictures that represent possible options.

palette menu

Palette that displays the top-level palette of pictures as well as subpalettes that represent possible options.

panel window

VI window that contains the front panel, the toolbar, and the icon/connector pane.

parallel mode

A type of SCXI operating mode in which the module sends each of its input channels directly to a separate analog input channel of the device to the module.

Pascal string (PStr)

A series of unsigned characters, with the value of the first character indicating the length of the string. Used in the Pascal programming language.

path

Description of the location of a file or directory, including the volume containing the file or directory, the directories between the top level and the file or directory, and the file or directory name.

pattern generation

A type of handshaked (latched) digital I/O in which internal counters generate the handshaked signal, which in turn initiates a digital transfer. Because counters output digital pulses at a constant rate, this means you can generate and retrieve patterns at a constant rate because the handshaked signal is produced at a constant rate.

PC

Personal computer.

peek

To read the contents.

PGIA

Programmable gain instrumentation amplifier.

pixmap

A 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 operating system.

plot

A graphical representation of an array of data shown either on a graph or chart.

Plug and Play devices

Devices that do not require dip switches or jumpers to configure resources on the devices--also called switchless devices.

pointer

A data structure that contains an address or other indication of storage location.

poke

An instruction that places a value into a specific location in memory.

polymorphism

Ability of a node to automatically adjust to data of different representation, type, or structure.

pop up

To call up a special menu by clicking on an object with the right mouse button (Windows, Sun and HP-UX) or holding down the <command> key while clicking (Macintosh).

pop-up menus

Menus accessed by popping up on an object. Menu options pertain to that object specifically.

portable

Able to compile on any platform that supports LabVIEW.

Positioning tool

Tool used to move and resize objects.

postriggering

The technique you use on a data acquisition board to acquire a programmed number of samples after trigger conditions are met.

PPC

Program-to-Program Communication. A low-level form of IAC by which applications send and receive blocks of data.

The GPIB command that configures an addressed Listener to participate in polls.

pretriggering

The technique you use on a data acquisition board to keep a continuous buffer filled with data, so that when the trigger conditions are met, the sample includes the data leading up to the trigger condition.

private data structures

Data structures whose exact format is not described and is usually subject to change.

privileged access

See supervisory access.

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.

protocol

Set of rules or conventions that cover the exchange of information between computer systems.

pseudocode

Simplified language-independent representation of programming code.

pull-down menus

Menus accessed from a menu bar. Menu options are usually general in nature.

pulse trains

Multiple pulses.

pulsed output

A form of counter signal generation by which a pulse is outputted when a counter reaches a certain value.

ATN

The GPIB attention line that distinguishes between commands and data messages. When ATN is asserted, bytes on the GPIB DIO lines are commands.

CIC

See Controller-In-Charge.

commands or command messages

GPIB message that performs tasks such as initializing the bus, addressing and unaddressing devices, and setting device modes.

Controller or Controller-In-Charge

The device that manages the GPIB by sending interface messages to other devices. The GPIB itself can also act as Controller-In-Charge.

CTS

Clear To Send (line).

data or data messages

Device-dependent messages that contain programming instructions, measurement results, machine status, and so on.

DAV

Data valid. One of the three GPIB handshake lines. *See* handshake.

device function

A function that operates on or otherwise pertains to a GPIB device rather than to the GPIB board in the computer.

DIO

Digital Input/Output (line).

DIO1 through DIO8

The GPIB lines that transmit command or data bytes from one device to another.

DSR

Data Set Ready (line).

DTR

Data Terminal Ready (line).

EABO

An error code that means the I/O operation aborted.

ECIC

An error code that means the function requires the GPIB board to be Controller-In-Charge.

EOI

End or identify. A GPIB line that is used to signal either the last byte of a data message (END) or carry the parallel poll identify (IDY) message.

ESAC

An error code which means the GPIB board is not System Controller as required.

ETAB

An error code which means there is a table problem.

General Purpose Interface Bus

See GPIB.

GPIB address

The address of a device on the GPIB, composed of a primary address (MLA and MTA) and perhaps a secondary address (MSA). The GPIB board has both a GPIB address and an I/O address.

GPIB board

Reference to the National Instruments family of GPIB boards.

GTL

Go to local. The GPIB command used to place an addressed Listener in local (front panel) control mode.

handshake

The mechanism used to transfer bytes from the source handshake function of one device to the acceptor handshake function of another device. The three GPIB lines DAV, NRFD, and NDAC are used in an interlocked fashion to signal the phases of the transfer, so that bytes can be sent asynchronously (for example, without a clock) at the speed of the slowest device.

high-level function

A device function that combines several rudimentary board operations into one function so that the user does not have to be concerned with bus management or other GPIB protocol matters. *See* low-level function.

IDY

Mnemonic for *identify*, a remote GPIB message.

IFC

Mnemonic for *interface clear*, a remote GPIB message.

interface message

A broadcast message sent from the Controller to all devices and used to manage the GPIB. Common interface messages include Interface Clear, listen addresses, talk addresses, and Serial Poll Enable/Disable. *See* data or data messages.

ISO

International Standards Organization.

ist

An individual status bit of the status byte used in the parallel poll configure function.

Listener

A GPIB device that receives data messages from a Talker.

LLO

Local lockout. The GPIB command used to tell all devices that they can or should ignore remote (GPIB) data messages or local (front panel) controls, depending on whether the device is in local or remote program mode.

LOK

Lockout bit.

low-level function

A rudimentary board or device function that performs a single operation.

m

Meters.

MLA

My Listen Address.

MSA

My Secondary Address.

MTA

My Talk Address.

NDAC

Not data accepted. One of the three GPIB handshake lines. *See* handshake.

NL

New line or linefeed.

NRFD

Not ready for data. One of the three GPIB handshake lines. *See* handshake.

oct

Octal.

parallel poll configure

The process of polling all configured devices at once and reading a composite poll response. *See* serial poll. Also, *see* PPC.

parallel poll disable

See PPD.

parallel poll enable

See PPE.

parallel poll unconfigure

See PPU.

PPD

The GPIB command that disables a configured device from participating in polls. There are 16 PPD commands.

PPE

The GPIB command that enables a configured device to participate in polls and to assign a DIO response line. There are 16 PPE commands.

PPU

The GPIB command that disables any device from participating in polls.

REN

Remote enable. A GPIB line controlled by the System Controller but used by the CIC to place devices in remote program mode.

RTS

Request To Send (line).

SCPI

Standard Commands for Programmable Instruments.

serial poll

The process of polling and reading the status byte of one device at a time. *See* parallel poll.

SRQ

Service request. The GPIB line that a device asserts to notify the CIC that the device needs servicing.

status byte

The data byte sent by a device when it is serially polled.

System Controller

The single designated Controller that can assert control (become CIC of the GPIB) by sending the Interface Clear (IFC) message. Other devices can become CIC only by having control passed to them.

Talker

A GPIB device that sends data messages to Listeners.

UNL

Unlisten. The GPIB command that unaddresses any active Listeners.

UNT

Untalk The GPIB command that unaddresses an active Talker.

VISA

Virtual Instrument Software Architecture. A single interface library for controlling VXI, GPIB, RS-232, and other types of instruments.

query

Like a *command*, causes a device to take some action, but requires a response containing data or other information. A command does not require a response.

queue

A group of items waiting to be acted upon by the computer. The arrangement of the items determines their processing priority. Queues are usually accessed in a FIFO fashion.

RAM

Random access memory.

read

To get information from any input device or file storage media.

read mark

Points to the scan at which a read operation begins. Analogous to a file I/O pointer, the read mark moves every time you read data from an input buffer. After the read is finished, the read mark points to the next unread scan. Because multiple buffers are possible, you need both the buffer number and the scan number to express the position of the read mark.

read mode

Indicates one of the four reference marks within an input buffer that provides the reference point for the read. This reference can be the read mark, the beginning of the buffer, the most recently acquired data, or the trigger position.

recursive filter

See IIR filter.

reentrant execution

Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.

reference

See pointer.

referenced signal sources

Signal sources with voltage signals that are referenced to a system ground, such as the earth or a building ground. Also called grounded signal sources.

refnum

An identifier of a DDE conversation or open files that can be referenced by related VIs.

register

A high-speed device used in a CPU for temporary storage of small amounts of data or intermediate results during processing.

Register-Based device

A Servant-only device that supports only the four basic VXIbus configuration registers. Register-Based devices are typically controlled by Message-Based devices via device-dependent register reads and writes.

regression analysis

See curve fitting.

relocatable

Able to be moved by the memory manager to a new memory location.

remote address

Address of the remote machine associated with a connection.

representation

Subtype of the numeric data type, of which there are signed and unsigned byte, word, and long integers, as well as single-, double-, and extended-precision floating-point numbers.

REQF

Request False; a VXI Event condition transferred using either VXI signals or VXI interrupts, indicating that a Servant no longer has a need for service.

REQT

Request True; a VXI Event condition transferred using either VXI signals or VXI interrupts, indicating that a Servant has a need for service.

resizing handles

Angled handles on the corner of objects that indicate resizing points.

Resource Manager

A Message-Based Commander located at Logical Address 0, which provides configuration management services such as address map configuration, Commander and Servant mappings, and self-test and diagnostic management.

response signal

Used to report changes in Word Serial communication status between a Servant and its Commander.

ring control

Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

ripple

A measure of the deviation of a filter from the ideal filter specifications.

referenced single-ended (RSE) measurement system

All measurements are made with respect to a common reference or a ground. Also called a grounded measurement system.

RM

See Resource Manager.

RMS

Root mean square.

ROAK

Release On Acknowledge; a type of VXI interrupter which always deasserts its interrupt line in response to an IACcycle on the VXIbus. All Message-Based VXI interrupters must be ROAinterrupters.

ROR

Release On Request; a type of VME bus arbitration where the current VMEbus master relinquishes control of the bus only when another bus master requests the VMEbus.

RORA

Release On Register Access; a type of VXI/VME interrupter which does not deassert its interrupt line in response to an IACcycle on the VXIbus. A device-specific register access is required to remove the interrupt condition from the VXIbus. The VXI specification recommends that VXI interrupters be only ROAinterrupters.

row-major order

A way to organize the data in a 2D array by rows.

RR

Read Ready; a bit in the Response register of a Message-Based device used in Word Serial Protocol indicating that a response to a previously sent query is pending.

RRviol

Read Ready protocol violation; a type of Word Serial Protocol error. If a Commander attempts to read a response from the Data Low register when the device is not Read Ready (does not have a response pending), a Read Ready violation may be generated.

RSE

Referenced single-ended.

RTD

Resistance temperature detector. A temperature-sensing device whose resistance increases with increases in temperature.

RTSI

Real-Time System Integration bus. The National Instruments timing bus that interconnects data acquisition boards directly, by means of connectors on top of the boards, for precise synchronization of functions.

run_me.llb

A LabVIEW DAQ VI library containing VIs that perform basic operations concerning analog I/O, digital I/O, and counters.

sample

A single (one and only one) analog or digital input or output data point.

sample counter

The clock that counts the output of the channel clock, in other words, the number of samples taken. On boards with simultaneous sampling, this counter counts the output of the scan clock and hence the number of scans.

scalar

Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans, strings and clusters are explicitly singular instances of their respective data types.

scale

Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.

scan

One or more analog or digital input samples. Typically, the number of input samples in a scan is equal to the number of channels n the input group. For example, one pulse from the scan clock produces one scan which acquires one new sample from every analog input channel in the group.

scan clock

The clock controlling the time interval between scans. On boards with interval scanning support (for example, the AT-MIO-16F-5), this clock gates the channel clock on and off. On boards with simultaneous sampling (for example, the EISA-A2000), this clock clocks the track-and-hold circuitry.

scan rate

The number of times (or scans) per second that LabVIEW acquires data from channels. For example, at a scan rate of 10Hz, LabVIEW samples each channel in a group 10 times per second.

scan width

The number of channels in the channel list or number of ports in the port list you use to configure an analog or digital input group.

scope chart

Numeric indicator modeled on the operation of an oscilloscope.

Scroll tool

Tool used to scroll windows.

SCSI

Small Computer System Interface (bus).

SCXI

Signal Conditioning eXtensions for Instrumentation. The National Instruments product line for conditional low-level signals within an external chassis near sensors, so only high-level signals in a noisy environment are sent to data acquisition boards.

sec

Seconds.

SEMI-SYNC Protocol

A one-line, open collector, multiple-device handshake trigger protocol.

sequence local

Terminal used to pass data between the frames of a Sequence Structure.

Sequence Structure

Program control structure that executes its subdiagrams in numeric order. Commonly used to force nodes that are not data dependent to execute in a desired order.

Servant

A device controlled by a Commander.

server

The application that receives messages from the client application in a dynamic data exchange.

settling time

The amount of time required for a voltage to reach its final value within specified limits.

shared external subroutine

Subroutine that can be shared by several CIN code resources.

Shared Memory Protocol

A communications protocol for Message-Based devices that uses a block of memory that is accessible to both a client and a server. The memory block acts as the medium for the protocol transmission.

shift register

Optional mechanism in loop structures used to pass a variable's value from one iteration of a loop to a subsequent iteration.

short integer

Data type of 16 bits, same as *word*.

signal

Any communication between Message-Based devices consisting of a write to a Signal register. Sending a signal requires that the sending device have VMEbus master capability.

signal conditioning

The manipulation of signals to prepare them for digitizing.

signal divider

Performing frequency division on an external signal.

signed integer

n bit pattern, interpreted such that the range is from $-2^{(n-1)}$ to $+2^{(n-1)} -1$.

simple-buffered I/O

Input/output operation that uses a single memory buffer big enough for all of your data. LabVIEW transfers data into or out of this buffer at the specified rate, beginning at the start of the buffer and stopping at the end of the buffer. You use simple buffered I/O when you acquire small amounts of data relative to memory constraints.

single-ended inputs

Analog inputs that you measure with respect to a common ground.

sink terminal

Terminal that absorbs data. Also called a destination terminal.

slave

A functional part of a MXI/VME/VXIbus device that detects data transfer cycles initiated by a VMEbus master and responds to the transfers when the address specifies one of the device's registers.

slider

Moveable part of slide controls and indicators.

SMP

See Shared Memory Protocol.

software trigger

A programmed event that triggers an event such as data acquisition.

software triggering

A method of triggering in which you to simulate an analog trigger using software. Also called conditional retrieval.

source code

Original, uncompiled text code.

SOURCE input pin

An counter input pin where the counter counts the signal transitions.

source terminal

Terminal that emits data.

spreadsheet

Any of a number of programs that arrange data and formulas in a matrix of cells.

status/ID

A value returned during an IACcycle. In VME, usually an 8-bit value which is either a status/data value or a vector/ID value used by the processor to determine the source. In VXI, a 16-bit value used as a data; the lower 8 bits form the VXI logical address of the interrupting device and the upper 8 bits specify the reason for interrupting.

STC

System timing controller.

strain gauge

A thin conductor, which is attached to a material, that detects stress or vibrations in that material.

string

A connected sequence of characters or bits treated as a single data item.

string controls and indicators

Front panel objects used to manipulate and display or input and output text.

strip chart

A numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.

structure

Program control element, such as a Sequence, Case, For Loop, or While Loop.

STST

START/STOP trigger protocol; a one-line, multiple-device protocol which can be sourced only by the VXI Slot 0 device and sensed by any other device on the VXI backplane.

stub VI

A nonfunctional prototype of a subVI that is created by the user. It has inputs and outputs, but is incomplete. It is used during early planning stages of VI design as a place holder for future VI development.

subdiagram

Block diagram within the border of a structure.

subVI

VI used in the block diagram of another VI; comparable to a subroutine.

supervisory access

One of the defined types of VMEbus data transfers; indicated by certain address modifier codes.

SVD

Singular value decomposition.

sweep chart

Similar to a scope chart except a line sweeps across the screen to separate old data from new data.

switchless device

Devices that do not require dip switches or jumpers to configure resources on the devices--also called Plug and Play devices.

SYNC Protocol

The most basic trigger protocol-simply a pulse of a minimum duration on any one of the trigger lines.

synchronous communications

A communications system that follows the command/response cycle model. In this model, a device issues a command to another device; the second device executes the command and then returns a response. Synchronous commands are executed in the order they are received.

syntax

The set of rules to which statements must conform in a particular programming language.

SYSFAIL*

A VMEbus signal that is used by a device to indicate an internal failure. A failed device asserts this line. In VXI, a device that fails also clears its PASSed bit in its Status register.

SYSRESET*

A VMEbus signal that is used by a device to indicate a system reset or power-up condition.

system hierarchy

The tree structure of the Commander/Servant relationships of all devices in the system at a given time. In the VXIbus structure, each Servant has a Commander. A Commander can in turn be a Servant to another Commander.

table-driven execution

A method of execution in which individual tasks are separate cases in a Case structure that is embedded in a While Loop. are specified Sequences as arrays of case numbers.

task

A timed I/O operation using a particular group. *See* task ID.

task ID

A number generated by LabVIEW, which encodes the device number and group number after you configure a group. You use the group configuration VIs in each function group to create the task ID, which you use when you run other VIs to specify the boards or channels on which the VIs operate. For example, the task ID in hex for an analog input task that uses group 2 and device 3 is 01020003. The following table shows how the configuration VIs encode the bits of the task ID.

Bits	Purpose
31 through 28	Reserved
27 through 24	Function code
23 through 20	Reserved
19 through 16	Group number
15 through 0	Device number

The following table gives the function code definitions. .

Function Code	I/O Operation
1	analog input
2	analog output
3	digital port I/O
4	digital group I/O
5	counter/timer I/O

TC

Terminal count. The highest value of a counter.

TCP

Transmission Control Protocol. *See* TCP/IP.

TCP/IP

Transmission Control Protocol/Internet Protocol. A suite of communications protocols that you use to transfer blocks of data between applications.

terminal

Object or region on a node through which data passes.

TIC

Trigger Interface Chip; a proprietary National Instruments ASIC used for direct access to the VXI trigger lines. The TIC contains a 16-bit counter, a dual 5-bit tick timer, and a full crosspoint switch.

tick

The smallest unit of time as measured by an operating system.

timeout

The time (in milliseconds) that a VI waits for an operation to complete. Generally, a timeout of -1 causes a VI to wait indefinitely. This is a feature of LabVIEW that prevents I/O functions from hanging indefinitely when there is a problem on the GPIB.

toggled output

A form of counter signal generation by which the output changes the state of the output signal from high to low, or low to high when the counter reaches a certain value.

tool

Special LabVIEW cursors with which you can perform specific operations.

toolbar

Bar containing command buttons that you can use to run and debug VIs.

Tools palette

Palette containing tools you can use to edit and debug front panel and block diagram objects.

top-level VI

VI at the top of the VI hierarchy. This term is used to distinguish the VI from its subVIs.

track-and-hold

A circuit that tracks an analog voltage and holds the value on command.

transducer excitation

A type of signal conditioning that uses external voltages and currents to excite the circuitry of a signal conditioning system into measuring physical phenomena.

trigger

A condition for starting or stopping clocks.

tristated

Defines logic that can have one of three states: low, high, and high-impedance.

TTL

Transistor-Transistor Logic

tunnel

Data entry or exit terminal on a structure.

type descriptor

See data type descriptor.

UART

Universal Asynchronous Receiver Transmitter

UDP

User Datagram Protocol. *See* TCP/IP.

unasserted

A signal in its inactive false state.

unipolar

A signal range that is either always positive or negative, but never both (for example 0 to 10 V, not -10 to 10 V).

universal constant

Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, for example, pi.

unsigned integer

n bit pattern interpreted such that the range is from 0 to 2^n -1.

UnSupCom

Unsupported Command; a type of Word Serial Protocol error. If a Commander sends a command or query to a Servant which the Servant does not know how to interpret, an Unsupported Command protocol error is generated.

update

One or more analog or digital output samples. Typically, the number of output samples in an update is equal to the number of channels in the output group. For example, one pulse from the update clock produces one update which sends one new sample to every analog output channel in the group.

update rate

The number of output updates per second.

update width

The number of channels in the channel list or number of ports in the port list you use to configure an analog or digital output group.

user-defined constant

Block diagram object that emits a value you set.

UT

Universal Time.

utility

A program that helps the user run, enhance, create, or analyze other programs and systems.

UUT

Unit under test.

V

volts.

VDC

Volts, direct current.

vector

One-dimensional array.

VI

Virtual instrument. A LabVIEW program; so called because it models the appearance and function of a physical instrument.

VIC

VXI Interactive Control program, a part of the NI-VXI bus interface software package. Used to program VXI devices, and develop and debug VXI application programs. Called *VICtext* when used on text-based platforms.

virtual instrument (VI)

LabVIEW program; so called because it models the appearance and function of a physical instrument.

VME

Versa Module Eurocard or IEEE 1014

VMEbus Class device

Also called non-VXIbus or foreign devices when found in VXIbus systems. They lack the configuration registers required to make them VXIbus devices.

Vref

Voltage reference.

VXIbus

VMEbus Extensions for Instrumentation

vxiedit

VXI Resource Editor program, a part of the NI-VXI bus interface software package. Used to configure the system, edit the manufacturer name and ID numbers, edit the model names of VXI and non-VXI devices in the system, as well as the system interrupt configuration information, and display the system configuration information generated by the Resource Manager. Called *vxitedit* when used on text-based platforms.

waveform

Multiple voltage readings taken at a specific sampling rate.

WDAQCONF.EXE

The NI-DAQ configuration utility in Windows.

While Loop

Post-iterative-test loop structure that repeats a section of code until a condition is met. Comparable to a Do loop or a Repeat-Until loop in conventional programming languages.

windowing FIR filter design

Method of designing linear-phase FIR filters in which you start with an ideal frequency response, calculate its impulse response, and then truncate the impulse response to produce a finite number of coefficients.

wire

Data path between nodes.

Wiring tool

Tool used to define data paths between source and sink terminals.

word

A data quantity consisting of 16 bits.

Word Serial Protocol

The simplest required communication protocol supported by Message-Based devices in the VXIbus system. It utilizes the A16 communication registers to perform 16-bit data transfers using a simple polling handshake method.

WR

Write Ready; a bit in the Response register of a Message-Based device used in Word Serial Protocol indicating the ability for a Servant to receive a single command/query written to its Data Low register.

write

Copying data to a storage device.

write mark

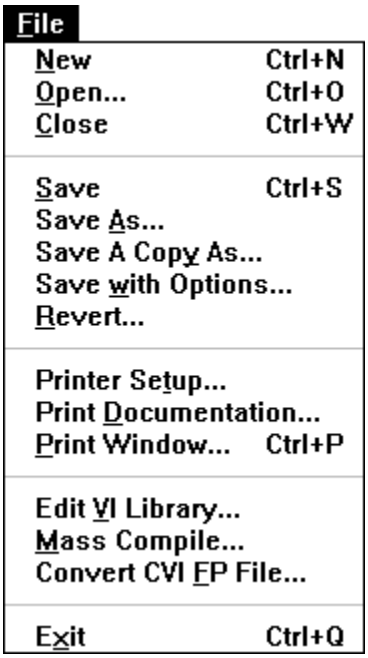
Points to the update at which a write operation begins. Analogous to a file I/O pointer, the write mark moves every time you write data into an output buffer. After the write is finished, the write mark points to the next update to be written. Because multiple buffers are possible, you need both the buffer number and the update number to express the position of the write mark.

WRviol

Write Ready protocol violation; a type of Word Serial Protocol error. If a Commander attempts to write a command or query to a Servant that is not Write Ready (already has a command or query pending), a Write Ready protocol violation may be generated.

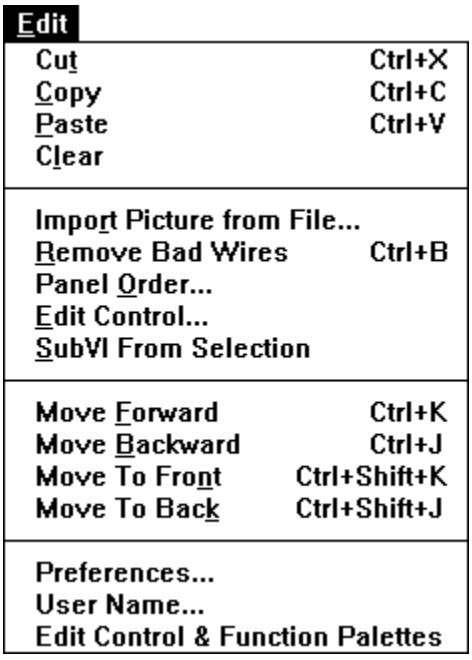
File Menu

You use the options in the **File** menu primarily to open, close, save, and print VIs.



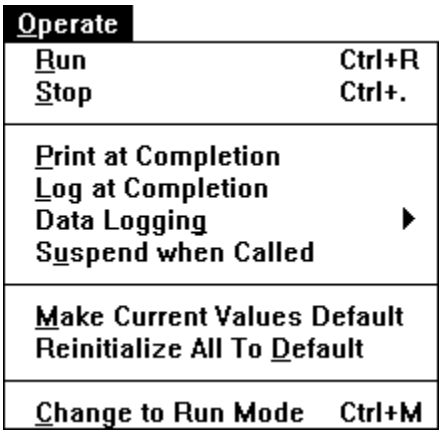
Edit Menu

You use the options in the **Edit** menu to build the front panel and block diagram of a VI.



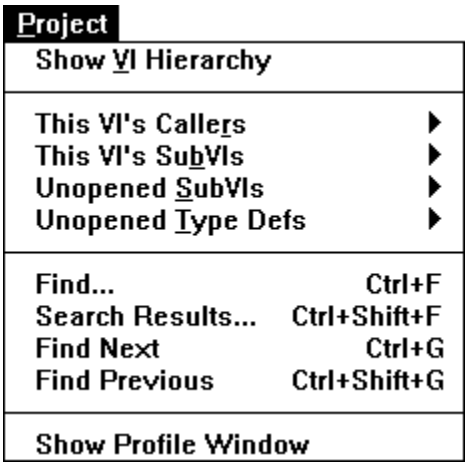
Operate Menu

You use the commands in the **Operate** menu to execute the VI.



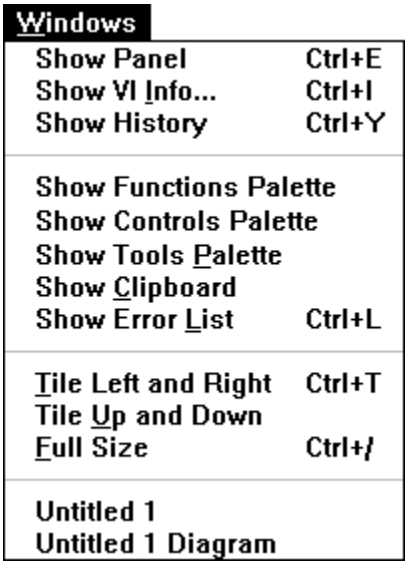
Project Menu

The **Project** menu has options to determine the hierarchy of your VI, to find objects or text in various VIs, and to determine the performance of your VI.



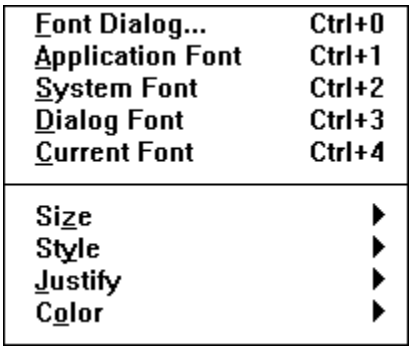
Windows Menu

You use the **Windows** menu to locate opened windows quickly and to open windows of subVIs and calling VIs.



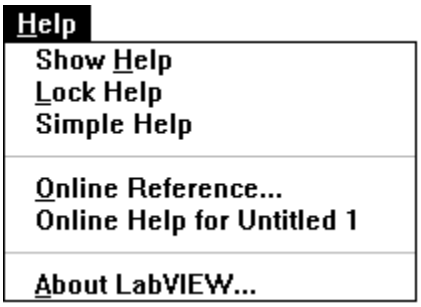
Text Menu

You use the **Text** menu to change the font, style, and color of text.



Help Menu

This menu provides you with online information about LabVIEW



more

New

Creates a new VI and opens its panel.

Open...

Opens an existing VI.

Close

Closes the active window.

Save: Saves the current VI.
Save As...: Saves the current VI under a new name.
Save A Copy As...: Saves a copy of the current VI under a new name.
Save with Options...: Options for saving VIs without the hierarchy and/or block diagrams.

Saving VIs

Revert...

Reverts the current VI to the last saved version.

Printer Setup...

Sets configuration options for the printer.

more

Data Logging

Displays data logging options.

Data Logging in the Front Panel
Enable Database Access
Retrieving Data Using File I/O Functions

Get Info...

Displays a dialog box containing information on the current VI.

more

Edit VI Library...

Removes VIs in a library or rearranges VI palette order.

Editing the Contents of Libraries

Mass Compile...

Compiles all VIs in a library.

Import Picture...

Imports graphics files.

more

Build Application...

Builds a stand-alone application using Application Builder.

Print Documentation

You can make a more comprehensive printout of a VI, including information about the front panel, block diagrams, subVIs, controls, VI history, and so on, by selecting the **Print Documentation** option.

more

Print Window

You can use the **Print Window** option to make a quick printout of the current window.

more

Exit

Quits LabVIEW.

Cut

Removes the selected object and places it on the Clipboard.

Copy

Copies the selected object and places it on the Clipboard.

Paste

Places a copy of the Clipboard contents in the active window.

Clear

Deletes the selected object.

Remove Bad Wires

Deletes all faulty wiring connections.

Panel Order...

Changes the order number for front panel objects interactively..

more

Edit Control...

Invokes the Control Editor.

more

Alignment

Aligns selected items using a selected Alignment axis.

more

Align

Aligns the selected items by the current Alignment setting.

more

Distribution

Distributes selected items using a selected Distribution setting.

more

Distribute

Spaces the selected items by the current Distribution.

more

Move Forward

Moves selected item one position higher in the stack.

more

Move Backward

Moves the selected item one position lower in the stack.

more

Move to Front

Moves the selected item to the top of the stack.

more

Move To Back

Moves the selected item to the bottom of the stack.

more

Preferences

Sets preferences for memory, disk, and display.

Setting LabVIEW Preferences

Run

Executes the current VI.

more

Stop

Stops execution of the current VI.

more

Change to Run Mode

Toggles between Run mode and Edit mode.

Make Current Values Default

Makes current values the default on all controls and indicators.

Reinitialize All To Default

Sets all controls and indicators to their default values.

Show Diagram

Makes the diagram window active.

Show History

Displays the development history of a VI and records any changes made.

more

Show Error List

Displays any errors in a selected VI.

Show Clipboard

Displays the contents of the clipboard.

Show VI Hierarchy

Graphically displays the calling hierarchy of all VIs in memory.

more

Tile Left And Right

Displays Panel and Diagram windows side by side.

Tile Up And Down

Displays Panel and Diagram windows top to bottom.

Full Size

Uses the entire screen to display the active window.

This VIs Callers

Displays a palette of VIs that calls the current VI.

This VI's SubVIs

Displays a palette of subVIs of the current VI.

Unopened SubVIs

Displays a palette of subVIs that are unopened.

Unopened Type Defs

Displays a palette of Type Definitions that are unopened.

Untitled 1

Lists all the Panel windows currently open. The check mark indicates the active window.

Untitled 1 Diagram

Lists all the Diagram windows currently open.

Apply Font

Displays options to let you choose from predefined font styles.

Font

Displays fonts available.

Size

Displays font sizes.

Style

Displays font styles such as plain, bold, italic, and underline.

Justify

Displays options to justify text such as left, center, and right.

Color

Displays a color palette to color text.

Convert CVI FP File (not available on the Macintosh)

Automates the process of convertings instrument drivers written in LabWindows/CVI so they can be used in LabVIEW. See the LabWindows/CVI Function Panel Converter topic for more information.

Import Picture from File

Imports a graphics file into the LabVIEW Clipboard.

SubVI from Selection

Converts a portion of a VI into a subVI that can be called from another VI. Using this command, select a section of a VI, and that section becomes a subVI. For more information see the Creating a SubVI from a Selection topic.

Edit User Name

This command is used to change the user name.

Front Panel Data Logging

Show Functions Palette

Displays block diagram functions in floating palette.

Show Controls Palette

Displays front panel controls in floating palette

Show Tools Palette

Displays editing and debugging tools in floating palette.

Show Panel

Makes the front panel the active window.

Show VI Information

Displays the VI file path, revision number, and memory usage.

Edit Control & Function Palettes

Customizes the controls and VIs in the **Controls** and **Functions** palettes. You can add your own controls and VIs to the palettes as well as rearranging the built-in palettes to make frequently-used functions more accessible.

Customizing the Controls and Functions Palettes

Log at Completion

Logs a time stamp and the data in all the front panel controls of a VI to a separate datalog file.

Suspend When Called

Suspends the execution of a VI when it is called by another VI (caller). While the execution of the VI is suspended, you can edit controls or execute the VI a number of times before returning to the caller.

Print at Completion

Prints the contents of a VI's front panel after each execution.

Font Dialog...

Displays a dialog box where you can edit the application, system, dialog, and current font.

Application Font

The LabVIEW default font used in the **Controls** and **Functions** palettes as well as the text in new controls.

System Font

The font LabVIEW uses for menus.

Dialog Font

The font LabVIEW uses for text in dialog boxes.

Current Font

Refers to the last font style selected.

Find

Finds occurrences of a particular object or string of text within an application consisting of multiple subVIs or a single large VI.

Using the Find Dialog Box
Using Find Options from a Pop-Up Menu

Search Results

Displays all search resluts.

Search Results Window

Find Next

Highlights the next result in the search list.

more

Find Previous

Highlights the previous result in the search list.

more

Show Profile Window

Shows an interactive tabular display of time and memory usage for each VI in your system.

Performance Profiling

Show Help

Context-sensitive Help window displays a function's or VI's parameters, parameter type definitions, and the description for the object.

Lock Help

If selected, prevents the contents of the Help window from changing as you move your mouse over other objects.

Simple Help

If selected, only displays required and recommended function parameters in Help window. If not selected, displays all function parameters.

Online Reference

Opens the online reference viewer and displays the contents page.

Online Help

Opens the LabVIEW Help window and displays the information for the current VI or function.

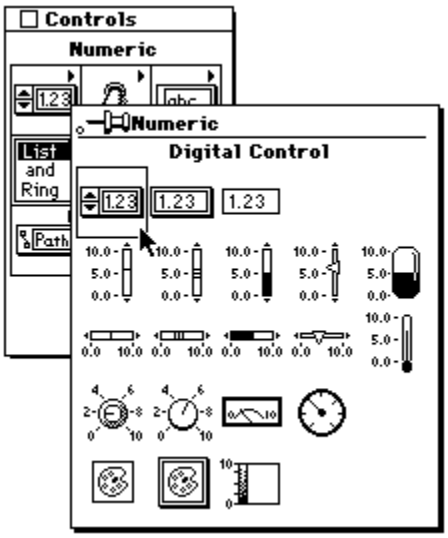
About LabVIEW...

Information on software version number and serial number.

Numeric Controls and Indicators

This topic explains how to create, operate, and configure the various styles of numeric controls and indicators.

When you select **Numeric** from the **Controls** palette, a palette of controls and indicators appears, as shown in the following illustration.



Numeric controls and indicators are either digital, slide, rotary, ring, enumerated, color box, or color ramp controls.

Digital Controls and Indicators
Slide Numeric Controls and Indicators
Rotary Numeric Controls and Indicatorsc
Color Box
Color Ramp
Units

Digital Controls and Indicators

A digital numeric control and indicator are shown in the following illustration.

Digital numerics are the simplest way to enter and display numeric data in LabVIEW.


Operating tool Use the Operating tool and any of the following methods to increment or decrement the displayed value:

- Click inside the digital display window and enter numbers from the keyboard.
- Click on the increment or decrement arrow of the digital control or indicator.
- Place the cursor to the right of the digit you want to change and press the up or down keyboard arrow.

enter button The enter button appears on the **Tools** palette to remind you that the new value replaces the old only when you press the `<Enter>` key on the numeric keypad, click outside the display window, or click the enter button. While the VI is running, this prevents LabVIEW from interpreting intermediate values as input. For example, while changing a value in the digital display to `135`, you do not want the VI to receive the values `1` and `13` before getting `135`. This rule does not apply to values you change using the increment/decrement arrows.
Numeric controls accept only the following--decimal digits, a decimal point, +, -, uppercase or lowercase `e`, and the terms `Inf` (infinity) and `NaN` (not a number), and in absolute time format the following characters: /, :, and uppercase or lowercase a, m, p, and m. If you exceed the limit for the selected representation, LabVIEW coerces the number to the natural limit. For example, if you enter `1234` into a control set for byte integers, LabVIEW coerces the number to `127`. If you incorrectly enter non-numeric values such as `aNN` for `NaN`, or `Ifn` for `Inf`, LabVIEW ignores them and uses the previous value.

The increment buttons usually change the ones digit. Incrementing a digit beyond `9` or decrementing below `0` affects the appropriate adjacent digit. Incrementing and decrementing repeats automatically. If you click on an increment button and hold the mouse button down, the display increments or decrements repeatedly. If you hold the `<Shift>` key down while incrementing repeatedly, the size of the increment increases by successively higher orders of magnitude. For example, by ones, then by tens, then by hundreds, and so on. As the range limit approaches, the increment decreases by orders of magnitude, slowing down to normal as the value reaches the limit.

In a digital control for time and date, individual components (second, minute, hour, day, month, year) as well as individual relative time components can be incremented and decremented.

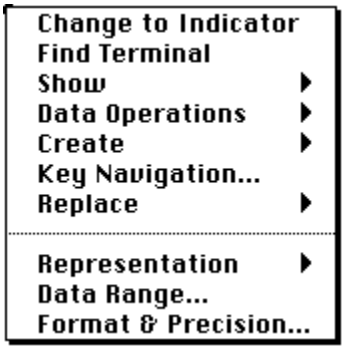
To increment a digital display by a digit other than the ones digit, use the Operating tool to place the insertion point to the right of the target digit. When you are finished, the ones digit again becomes the increment digit.

Numbers may become too large to fit in the digital display on the control. You can view the complete value by resizing the control, making it longer horizontally.

Digital Numeric Options
Integers Display in Other Radixes
Numeric Value Representation Changes
Range Options for Numeric Controls and Indicators
Numeric Range Checking
Format and Precision Changes of Digital Displays

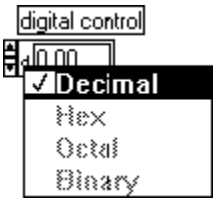
Digital Numeric Options

You can change the defaults for digital numerics through their pop-up menus. The pop-up menu for a digital numeric is shown in the following illustration.

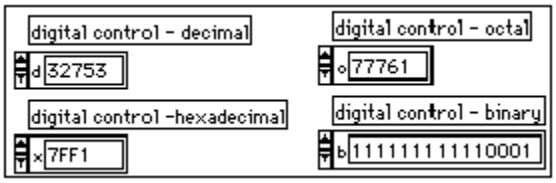


Integers Display in Other Radixes

You can display signed or unsigned integer data in hexadecimal, octal, and binary form, in addition to decimal form. To change the form, select **Show**»**Radix** from the numeric pop-up menu. A d appears on the housing of the numeric display as shown in the following illustration.



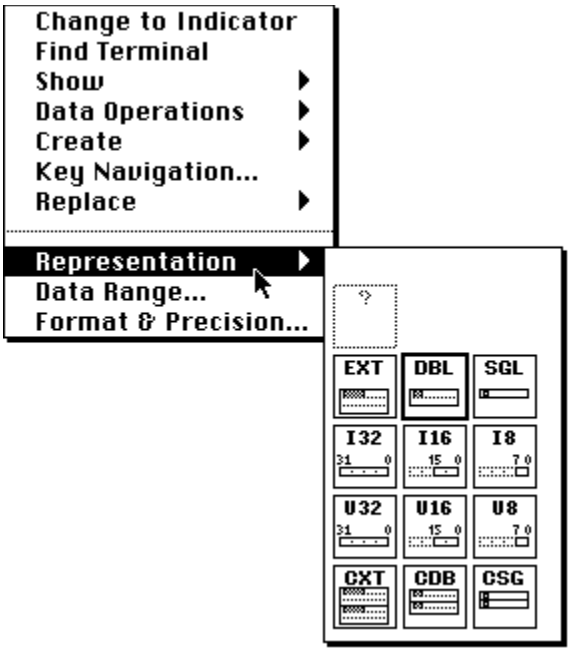
If you click on the d, the menu shown in the previous illustration appears. The number 32,753 is displayed in each radix in the following illustration.



Numeric Value Representation Changes

You can choose from 12 representations for a digital numeric control or indicator. Use the **Representation** option from the control or indicator pop-up menu to change to 32-bit single-precision (SGL), 64-bit double precision (DBL), extended-precision (EXT) floating-point numbers, or one of the six integer representations: signed (I8) or unsigned (U8) byte (8-bit), signed (I16) or unsigned (U16) word (16-bit), or signed (I32) or unsigned (U32) long (32-bit) integers. You can also choose complex extended-precision (CXT), complex double-precision (CDB), or complex single-precision (CSG) floating-point numbers. These choices are shown in the following illustration.

You can click on the following representation icons to learn their names.



Range Options for Numeric Controls and Indicators

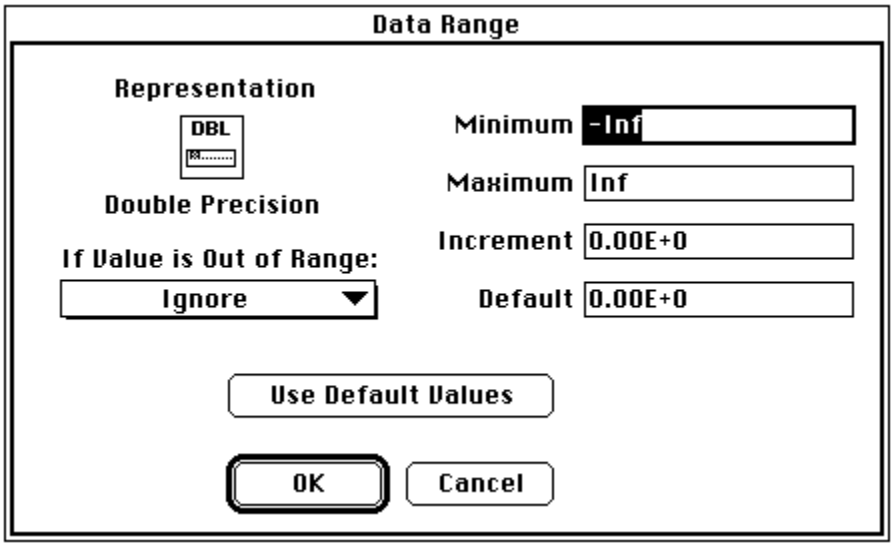
Each representation has natural minimum and maximum range limits. For example, signed byte integers are limited to values from -128 through 127, whereas floating-point numbers have the ranges shown in the following table.

Table 10-1. Range Options of Numeric Controls and Indicators

Precision	Single	Double	Extended (platform-dependent)
Maximum Positive Number	3.4E38	1.7E308	1.1E4932
Minimum Positive Number	1.5E-45	5.0E-324	1.9E-4951
Minimum Negative Number	-1.5E-45	-5.0E-324	-1.9E-4951
Maximum Negative Number	-3.4E38	-1.7E308	-1.1E4932

Note: Although LabVIEW can process in the range shown in Table 10-1, the range of extended floating-point numbers it can represent and display in text format is ±9.9999999999999999E999.

You can choose other limits within these natural bounds with the **Data Range...** option from the pop-up menu. The following dialog box appears.



Numeric Range Checking

You can also limit intermediate values to certain increments. For example, you might limit word integers to increments of 10, or single-precision floating-point numbers to increments of 0.25. If you change either the limits or the increment, you should also decide what to do if a VI or the operator attempts to set a value outside the range or off the increment. You have the following options:

Ignore	LabVIEW does not change or flag invalid values. Clicking on the increment and decrement arrows changes the value by the increment you set, but the value does not go beyond the minimum or maximum values.
Coerce	LabVIEW changes invalid values to the nearest valid value automatically. For example, if the minimum is 3, the maximum is 10, and the increment is 2, valid values are 3, 5, 7, 9, and 10. LabVIEW coerces the value 0 to 3, the value 6 to 7, and the value 100 to 10.
Suspend	LabVIEW suspends execution of a VI or subVI when a value is invalid. When you choose to suspend on invalid values, LabVIEW keeps a copy of all the front panel data in memory in case you need to open the front panel to show an error.

A top-level VI with controls whose values are invalid cannot run. If the value of a control is invalid before a subVI runs (when a subVI is about to execute), the VI is suspended. The front panel of the VI opens (or becomes the active window) and the invalid control(s) are outlined in red (or a thick black line on a black-and-white monitor). The toolbar on the block diagram of a suspended subVI looks like the following illustration.

You must set the control to a valid value before you can proceed. When all control values are valid, the toolbar looks like the following illustration, and you can click on the run button to continue execution.

If the value of an indicator or control becomes invalid while a subVI is running, execution pauses as if there were a breakpoint. The front panel of the VI opens (or becomes the active window) and the indicator(s) and control(s) that are currently invalid are outlined in red (or a thick black line on a black-and-white monitor). When a suspended VI finishes execution, if there are any invalid values, the return to

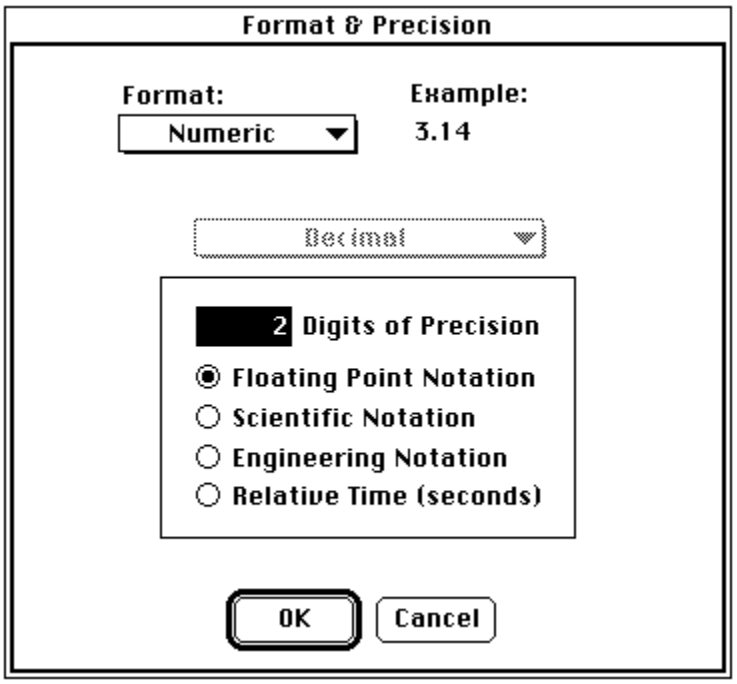
caller button is disabled, as shown in the following illustration.

You must set indicators to valid values before you can return to the calling VI. You can also change the control values to produce valid outputs and run the subVI again by clicking the run button. When all indicator values are valid, the toolbar looks like the following illustration, and you can click on the return to caller button to continue execution.

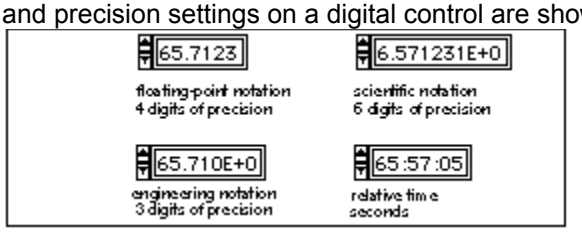
Format and Precision Changes of Digital Displays

You can select whether your digital displays are formatted for numerics or for time and date. If numeric, you can choose whether the notation is floating-point, scientific, engineering, or relative time in seconds, and you can select their precision, that is how many digits to the right of the decimal point they display, from 0 through 20. The precision you select affects only the display of the value; the internal accuracy still depends on the representation.

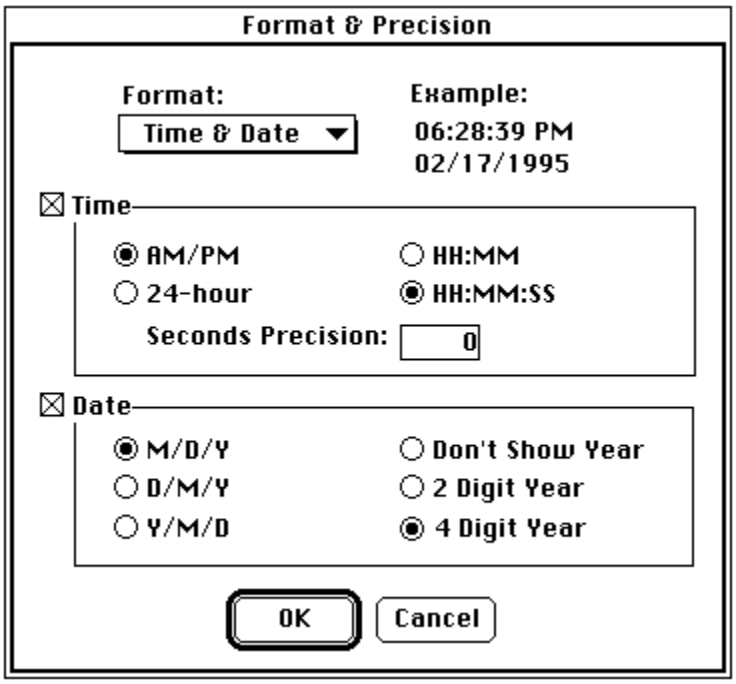
To change any of these parameters of the digital display, select **Format & Precision...** option from the display pop-up menu. The dialog box comes up in numeric format, as shown in the following illustration.



Notice that an example is shown at the top of the dialog box as you make selections. Examples of format and precision settings on a digital control are shown in the following illustration.



To format absolute time and/or date, select **Time & Date** from the **Format** pop-up menu at the top of the dialog box. The dialog box changes, as shown in the following illustration.



You can format for either time or date, or both. If you enter only time or only date, LabVIEW infers the unspecified components. If you do not enter time, LabVIEW assumes 12:00 a.m. If you do not enter date, it assumes the previous date value. If you enter date, but the control is not in a date format, LabVIEW assumes the month, day, and year ordering based on the settings in the Preferences dialog box. If you enter only two digits for the year, LabVIEW assumes the following: any number less than thirty-eight is in the twenty-first century, otherwise the number is in the twentieth century.

Though absolute time is displayed as a time and date string, it is represented internally as the number of seconds since 12:00 AM January 1, 1904, Greenwich Mean Time. LabVIEW keeps track of these components internally.

Note: When a control is in absolute time format, you always have the option to enter time, date, or time and date. If you do not want LabVIEW to assume a date, use relative time.

Notice the examples at the top right of the dialog box, which change as you make selections.

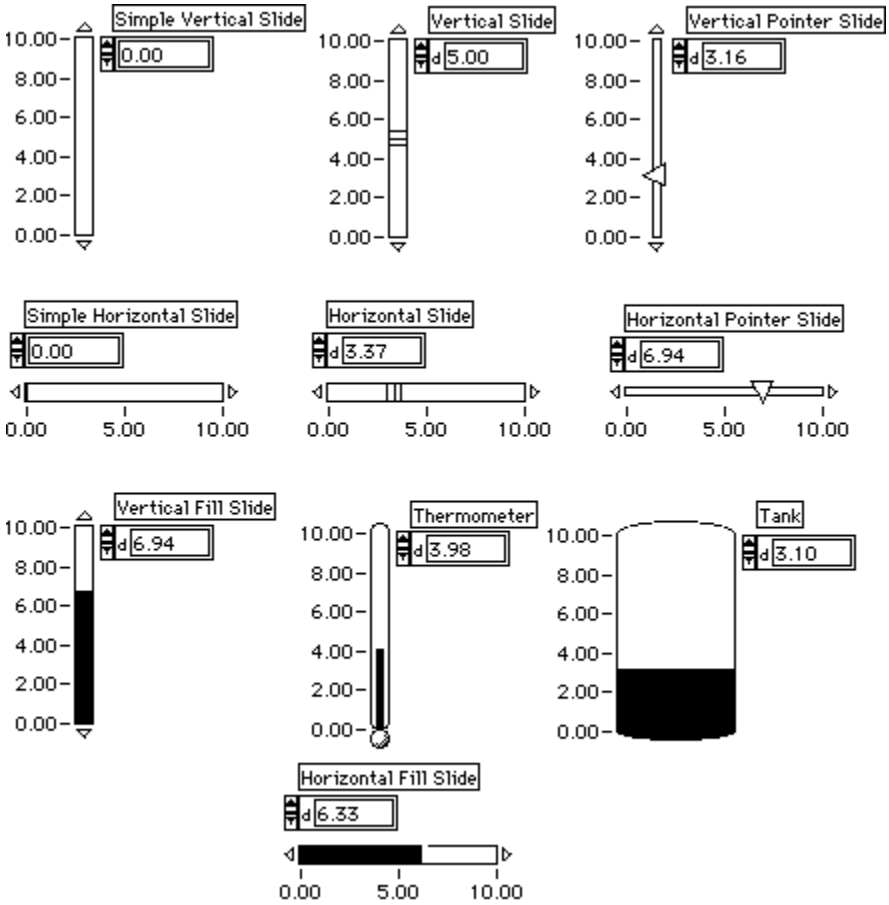
The valid range for time and date differs across computer platforms as follows:

- ***(Windows)*** 12:00 a.m. Jan. 2, 1970 - 12:00 a.m. Feb. 4, 2106
- ***(Windows NT)*** 12:00 a.m. Jan. 2, 1970 - 12:00 a.m. Jan. 17, 2038
- ***(Macintosh)*** 12:00 a.m. Jan. 2, 1904 - 12:00 a.m. Jan. 2, 2040
- ***(UNIX)*** 12:00 a.m. Dec. 15, 1901 - 12:00 a.m. Jan. 17, 2038

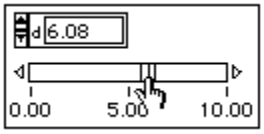
These ranges may be up to a few days wider depending on your time zone and whether daylight saving time is in effect.

Slide Numeric Controls and Indicators

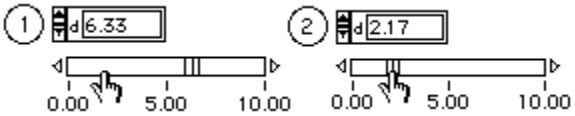
The slide controls and indicators are shown in the following illustration.



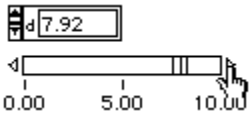
Each slide has a digital display. You can use the digital displays to enter data into slide controls, as explained in the Digital Controls and Indicators topic. You can use the Operating tool on such parts as the slider, the slide housing, the scale, and increment buttons to enter data or change values. The slider is the part that moves to show the control's value. The housing is the non-moving part that the slider moves on or over. The scale indicates the value of the slider, and the increment buttons are small triangles at either end of the housing. An example of a slider is shown in the following illustration.



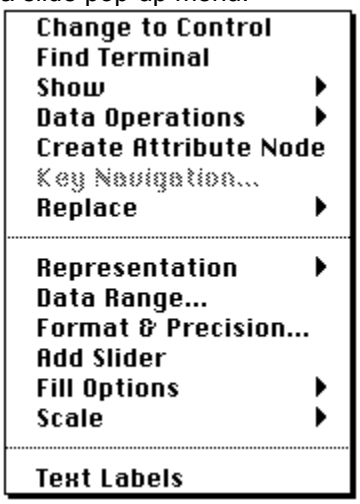
You can drag the slider with the Operating tool to a new position. If the VI is running during the change, intermediate values may pass to the program, depending on how often the VI reads the control.
You also can click on a point on the housing and the slider snaps to that location as shown in the following illustration. Intermediate values do not pass to the program.



If you use a slide that has increment buttons, you can click on an increment button, as shown following, and the slider moves slowly in the direction of the arrow. Hold down the <Shift> key to move the slider faster. Intermediate values may pass to the program.



Just like the digital numerics, slides have **Representation**, **Data Range...**, and **Format & Precision** options in their pop-up menus. These options work the same as they do for digital displays, except that slides cannot represent complex numbers. Slides also have other options. The following illustration shows a slide pop-up menu.



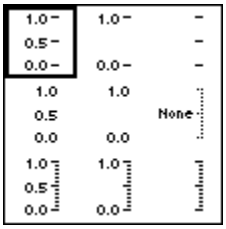
The **Show»Digital Display** option of the pop-up menu controls whether the slide's digital display appears.

Slide Scale
Text Scale
Filled and Multivalued Slides

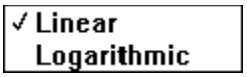
Slide Scale

The scale submenu options apply to the slide's scale only. The scale pop-up menu is shown in the following illustration.

The **Marker Spacing** option is discussed in the Specifying Uneven Scale Marker Distribution topic.
The **Format & Precision** option functions as described in the Digital Controls and Indicators topic.
The **Style** option gives you the palette shown in the following illustration. You can display a scale with no tick marks or no scale values, or you can hide the scale altogether.



The **Mapping** item gives you the option of linear or logarithmic scale spacing, as shown in the following illustration.



If you change to Logarithmic spacing, and the low scale limit is less than or equal to 0, the limit automatically becomes a positive number, and LabVIEW revalues other markers accordingly. Keep in mind that scale options, including the mapping functions, are independent of the slide data range values, which you change with the **Data Range** pop-up option. If you want to limit your data to logarithmic values, you need to change the data range to eliminate values less than or equal to zero.

Scale Marker Changes

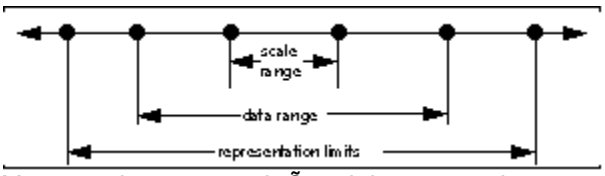
The scale of a numeric control or indicator has two or more markers, which are labels that show the value associated with the marker position.

You can change the outer two markers, the scale limits, and you can change any inner markers from even spacing to place them anywhere you want.

Scale Limit Changes
Specifying Uneven Scale Marker Distribution

Scale Limit Changes

The outer two markers are the scale limits, and they are not required to coincide with the range limits, but can be a subset of the range of the control or indicator. For example, the default range of a 16-bit signed integer control or indicator is -32,768 to 32,767. However, you can set the data range to be -1,000 to 1,000 and then set the scale limits to be 0 and 500.

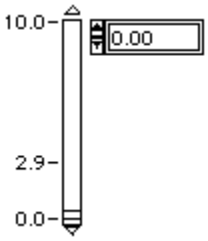


You can change a scaleÕs minimum, maximum, and increment interactively in five ways using either the Operating tool or the Labeling tool. If you want to change a scale programmatically, use the Attribute Node. For more information, see Attribute Nodes.

- If you type a new maximum value into its display, the minimum stays the same, and LabVIEW recalculates the increment automatically.
- If you type a new minimum value into its display, the maximum stays the same, and LabVIEW recalculates the increment automatically.
- If you type the current minimum value into the maximum display, LabVIEW flips the scale so that what was formerly the minimum is now the maximum, and vice versa. LabVIEW also recalculates the increments.
- If you type into any intermediate marker, the increment becomes that value minus the minimum.
- If you change the size of the slide, the increment adjusts so that the markers do not overlap.

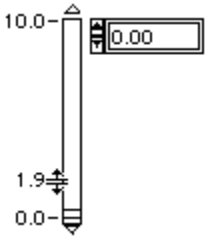
Specifying Uneven Scale Marker Distribution

By default, scale markers are evenly spaced. If you want, you can specify exactly where inner markers for a scale are located. This is useful for marking a few specific points on a slide (such as a set point or threshold). If you want non-uniform marker distribution, select **Scale**»**Marker Spacing**»**Arbitrary** from the scale's pop-up menu. Then you can pop up on the slide and select **Scale**»**Add Marker** or pop up on the marker and select **Add Marker**. A marker appears at an arbitrary location, as shown in the following illustration.



To delete an arbitrary marker, pop up on the slide and select **Scale**»**Delete Marker** or pop up on the marker and select **Delete Marker**.

Once a marker is created, you can type a number in the marker; the location changes automatically to match the number. You can also move markers by dragging ticks, the small lines beside the scale's data points. To do so, idle the Operating tool over the ticks. The cursor changes to show that the ticks can be moved, as shown in the following illustration.



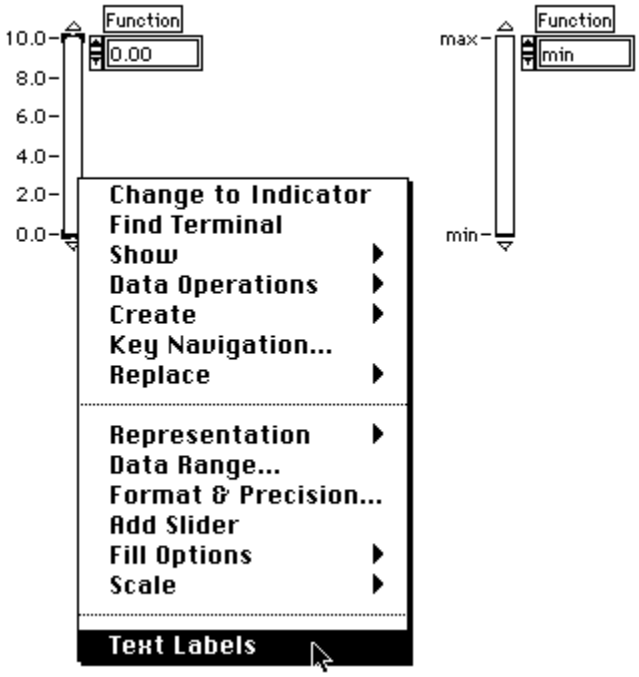
In Uniform mode, dragging a tick changes the marker distribution. In Arbitrary Markers mode, dragging a tick moves only the current marker, without changing the other markers. Pressing the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key while dragging creates a new marker. (If the ticks are hidden, as chosen in the **Scale**»**Style** dialog box, you cannot drag them.)

The arbitrary markers affect only the inner markers, not the two end markers. If no arbitrary markers are visible in the current range, the scale reverts temporarily to uniform markers.

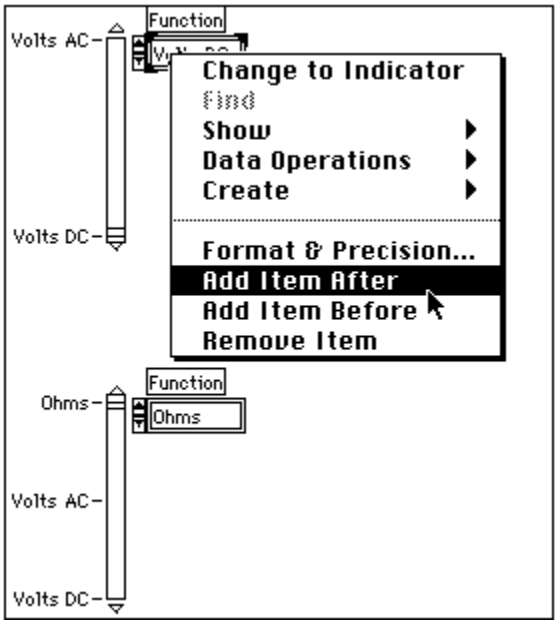
Text Scale

You can also use text labels on numeric scales. Labels are useful because they allow you to associate strings with integer values. This configuration is useful for selecting mutually exclusive options. To enable this option, select **Text Labels** from the slide pop-up menu. The slide control appears with default text labels, min and max, and you can begin typing immediately to enter the labels you want. Press <Shift-Enter> (Windows and HP-UX), or <Shift-Return> (Macintosh and Sun) after each label to create a new label, and <Enter> on the numeric keypad after your last label to finish.

You can use the Labeling tool to edit min and max labels in the text display or on the slide itself.



You can pop up on the text display and select **Show**»**Digital Display** to find out what numeric values are associated with the text labels you create. These values always start at zero (on the bottom of vertical slides and on the top of horizontal slides) and increase by one for each text label.
Use the **Add Item After** or **Add Item Before** option from the text display pop-up menu to create new labels, as shown in the following illustration.

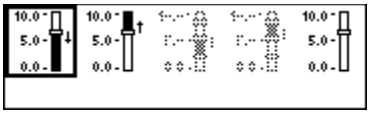


You can also press <Shift-Enter> (Windows and HP-UX), or <Shift-Return> (Macintosh and Sun) to advance to a new item when you are editing the existing items.

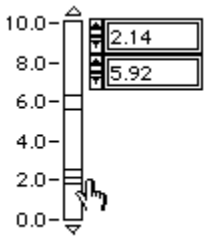
Filled and Multivalued Slides

The **Numeric** palette contains four controls that are configured to fill from the minimum to the slider value. These controls include a vertical slide, a horizontal slide, the tank, and the thermometer. Using the **Fill**

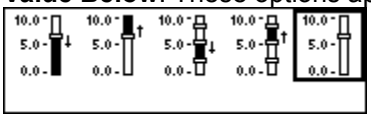
Options item in the pop-up menu, you can turn all slides into fill slide, and you can turn fill slides into regular slides. Normally, you have three choices, as shown in the following illustration. These are fill from the minimum value to the slider location, fill from the maximum value to the slider location, or use no fill.



You can also show more than one value on the same slide. To do so, choose **Add Slider** from the pop-up menu. A new slider appears along with a new digital display as shown in the following illustration.



When you do this, two more options appear in the **Fill Options** palette: **Fill to Value Above** and **Fill to Value Below**. These options apply to the active slider. All five options appear in the following illustration.

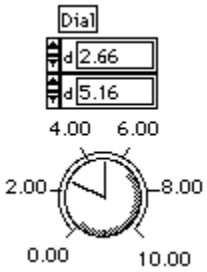


For an example of slide controls and indicators, see `examples\general\controls\alarmsld.llb`.

Rotary Numeric Controls and Indicators

The rotary numeric controls and indicators are shown in the following illustration

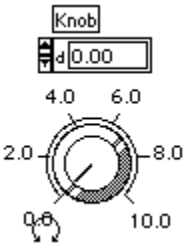
The rotary objects have most of the same options as the slide. The sliders (or needles) in rotary objects turn rather than slide, but you operate them the same way you operate slides.



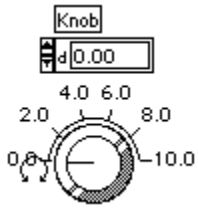
Rotary controls, like linear controls, can display more than one value, as shown in the preceding illustration. You add new values by selecting **Add Needle** from the pop-up menu, in the same way you add new values to slides.

If you move the Positioning tool to the scale, the tool changes to a rotary cursor. If you click and drag on the scale's outer limits, you can change the arc that the scale covers. If you click and drag on the scale's inner markers, you can rotate the arc; it still has the same range but different starting and ending angles. This procedure is shown in the following illustration. Hold down the `<Shift>` key to snap the arc to 45-degree angles.

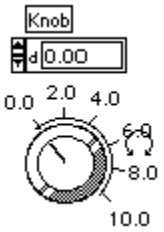
Place the cursor on the knob scale, where it changes appearance to a double arrow horseshoe.



Dragging outer markers changes the size of the scale arc.



Dragging inner markers rotates the arc.

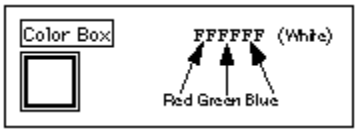


Tick marks of rotary scales can be dragged with the Operating tool, just like linear scales. Ticks can also be unevenly distributed in a similar manner to linear scales. See the Specifying Uneven Marker Distribution topic for more information.

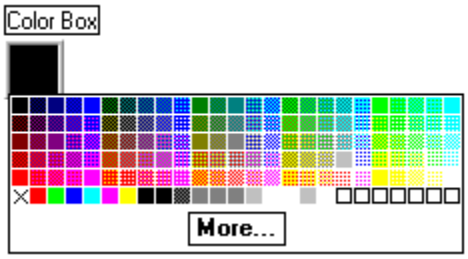
For more information on ring controls and indicators, see the List and Ring Controls and Indicators topic.

Color Box

The color box displays a color corresponding to a specified value. The color value is expressed as a hexadecimal number with the form `RRGGBB`, in which the first two numbers control the red color value, the second two numbers control the green color value, and the last two numbers control the blue color value. An example is shown in the following illustration.



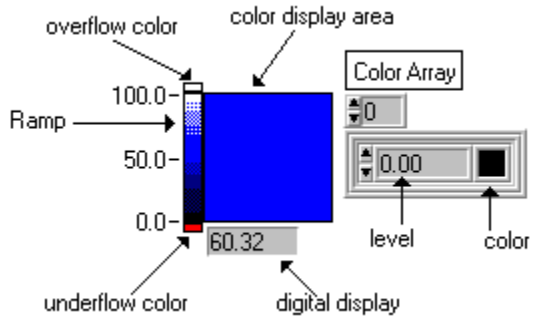
You can set the color of the color box by clicking on it with either the Operating or Color tool to display a Color palette, as shown in the following illustration. Releasing your mouse button on the color you want selects that color.



Color boxes are typically used on the front panel as indicators. The easiest way to set the color of a color box is to use the color box constant, from the **Functions**»**Numeric**»**Additional Numeric Constants** palette in the block diagram window. Wire the color box constant to the color box terminal on the block diagram. Clicking on the color box constant with either the Operating tool or the Color tool displays the same Color palette that you use to set the color box. If you want to change the color box indicator on the front panel to various colors which indicate different conditions, you can use a series of color box constants inside a Case Structure.
The small T in the Color palette represents the transparent color.

Color Ramp

The color ramp displays a color corresponding to a specified value. You can configure a color scale, which consists of at least two arbitrary markers, each consisting of a level and the display color corresponding to that level. As the input value changes, the color displayed changes to the color corresponding to that value. An example is shown in the following illustration.



You create these pairs using the arbitrary markers of the color scale. Each arbitrary marker specifies a level and has a color associated with it.
When the color ramp first appears on the front panel, it is set to have three levels. Level 0 is set to the value 0, and the color black. Level 1 is set to the value 50 and the color blue. Level 2 is set to the value 100 and the color white. You can add levels, change the values, and set the colors as you choose.
You can use the **Interpolate Colors** option of the color ramp pop-up menu to select whether the control interpolates colors to display shades of color for values between the specified levels, or changes to a specific color only when the input value reaches a level specified in the color array. If **Interpolate Colors** is off, the color is set to the color of the largest level less than or equal to the current value.
The color array is always sorted by level. The scale on the ramp corresponds to the largest and smallest values in the array. When the minimum or maximum of the scale changes, the color array levels are automatically redistributed between the new values.

The color scale has a number of options which you can display or hide. These options include the unit label, the digital display, and the ramp.

The ramp component of this control has an extra color at the top and the bottom of the scale. You can use these colors to select a color to display if an overflow or an underflow occurs. Click in these areas with the Operating tool and select your overflow and underflow colors from the Color palette.

Both the color ramp and the color box are used to display a numeric value as a color. With the color ramp, you can assign any range of numbers to any range of colors. The color box, however, displays only the color specified by the red, green, and blue components of the numeric value; any given value always maps to the same color.

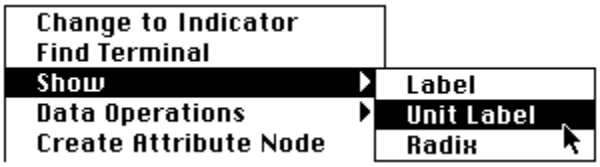
Note: The color ramp can have as many colors as your monitor can handle, but no more colors than what are available on your monitor.

You can use the color scale to specify color tables for the Intensity Graph and Intensity Chart controls. Graph and Chart Controls and Indicators for more information on these controls.

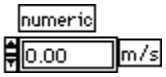
Units

Any numeric control in LabVIEW can have physical units, such as meters or kilometers/second, associated with it. Any numeric control with an associated unit is restricted to a floating point data type.

Units for a control are displayed and modified in a separate but attached label, called the unit label. You can show this label by selecting the option from the pop-up menu, as shown in the following illustration.



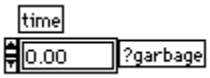
Once the unit label is displayed, you can enter a unit using standard abbreviations such as m for meters, `ft` for feet, `s` for seconds, and so on.
If you are not familiar with which units are allowed, enter a simple unit such as m, then pop up on the unit label and select **Unit...**. A dialog box appears that contains information about the units LabVIEW has available. You can use this dialog box to replace your first unit with a more appropriate choice.
The following numeric control is set to have units of meters per second.



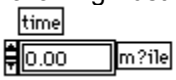
Click here to see all the units you can use with the units feature.
Entering Units
Units and Stricter Type Checking
Polymorphic Units

Entering Units

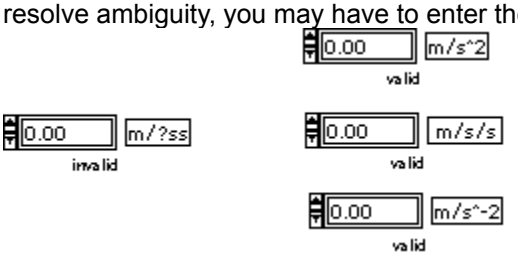
If you try to enter an invalid unit into a unit label, LabVIEW flags the invalid unit by placing a ? in the label, as shown in the following illustration.



Notice that you must enter units using the correct abbreviations or LabVIEW flags it, as shown in the following illustration.



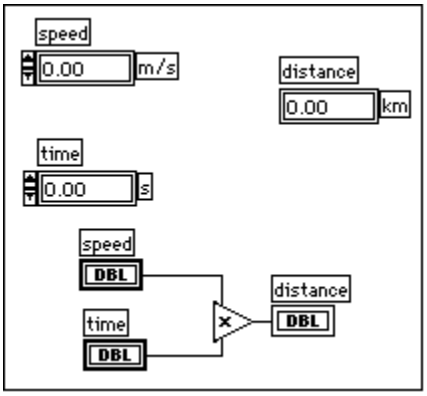
Also, notice that you cannot enter ambiguous units. Thus m/ss is flagged in the following illustration, because it is not clear whether it means meters per second squared, or (meters/seconds) * seconds. To resolve ambiguity, you may have to enter the units differently, as shown in the three examples to the right.



You cannot select units for a chart or graph unless you wire them to an object that has an associated unit. You can then show the unit that the chart or graph has acquired through the connection. You can change this unit to another unit that measures the same phenomenon, but not to a unit measuring a different class of quantity or quality. For example, if an input to a chart carries the unit mi, you could edit the chart unit to `ft`, `in`, or `m`, but not to `N`, `Hz`, `min`, `acre`, or `A`.

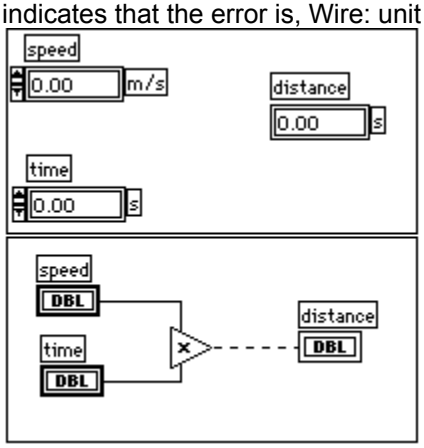
Units and Stricter Type Checking

A wire connected to a source that has a unit associated with it can only connect to a destination with a compatible unit, as shown in the following illustration.



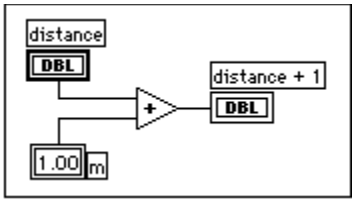
In the case of the previous illustration, the distance display is automatically scaled to display kilometers instead of meters, because that was the specified unit for the indicator.

Notice that you cannot connect signals with incompatible units, as shown in the following illustration. If you select **List Errors** from the pop-up menu for the broken wire shown below, the List Errors window indicates that the error is, Wire: unit conflict.



Some functions are ambiguous with respect to units and cannot be used with signals that have units. For example, the Add One function is ambiguous with respect to units. If you are using distance units, Add One cannot tell whether to add one meter, one kilometer, or one foot. Because of this ambiguity, the Add One function and similar functions cannot be used with data that has associated units.

One way around this difficulty is to use a numeric constant with the proper unit and the addition function on the block diagram to create your own Add One unit function.



Note: Units cannot be used in Formula Nodes.

Polymorphic Units

If you want to create a VI that computes the root mean square value of a waveform, you have to define the unit associated with the waveform. A separate VI would be needed for voltage waveforms, current

waveforms, temperature waveforms, and so on. To allow one VI to do the same calculation, regardless of the units received by the inputs, LabVIEW has polymorphic unit capability.

You create a polymorphic unit by entering `$x`, where *x* is a number (for example, `$1`). You can think of this as a placeholder for the actual unit. When the VI is called, LabVIEW substitutes the units you pass in for all occurrences of `$x` in that VI. The X place holder always gets passed a unit that is expressed in base units, for example, meters, meters/sec, not feet, inches, and so on.

A polymorphic unit is treated as a unique unit. It is not convertible to any other unit, and propagates throughout the diagram just as other units do. When it is connected to an indicator that also has the abbreviation `$1`, the units match and the VI can compile.

`$1` can be used in combinations like any other unit. For example, if the input is multiplied by 3 seconds and then wired to an indicator, the indicator must be `$1 s` units. If the indicator has different units, the block diagram shows a bad wire.

If you need to use more than one polymorphic unit, you can use the abbreviations `$2`, `$3`, and so on.

A call to a subVI containing polymorphic units computes output units based on the units received by its inputs. For example, suppose you create a VI that has two inputs with the polymorphic units `$1` and `$2` that creates an output in the form `$1 $2 / s`. If a call to the VI receives inputs with the unit m/s to the `$1` input and kg to the `$2` input, the output unit would be computed as `kg m / s^2`.

Suppose a different VI has two inputs of `$1` and `$1/s`, and computes an output of `$1^2`. If a call to this VI receives inputs of m/s to the `$1` input and `m/s^2` to the `$1/s` input, the output unit would be computed as `m^2 / s^2`. If this VI receives inputs of m to the `$1` input and kg to the `$1/s` input, however, one of the inputs would be declared to be a unit conflict and the output would be computed (if possible) from the other. A polymorphic VI can have a polymorphic subVI because the respective units are kept distinct.

Unit Names

Quantity Name	Unit	Abbreviation
Base Units		
plane angle	radian	rad
solid angle	steradian	sr
time	second	s
length	meter	m
mass	gram	g
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd
Derived Units with Special Names		
frequency	hertz	Hz
force	newton	N
pressure	pascal	Pa
energy	joule	J
power	watt	W
electric charge	coulomb	C
electric potential	volt	V
capacitance	farad	F
electric resistance	ohm	Ohm
conductance	siemens	S
magnetic flux	weber	Wb
magnetic flux density	tesla	T
inductance	henry	H
luminous flux	lumen	lm
illuminance	lux	lx

Celsius temperature	degree Celsius	degC
activity	becquerel	Bq
absorbed dose	gray	Gy
dose equivalent	sievert	Sv

Additional Units in Use with SI Units

time	minute	min
time	hour	h
time	day	d
plane angle	degree	deg
plane angle	minute	'
plane angle	second	"
volume	liter	l
mass	metric ton	t
area	hectare	ha
energy	electron volt	eV
mass	unified atomic mass unit	u

CGS Units

area	barn	b
force	dyne	dyn
energy	erg	erg
pressure	bar	bar

Other Units

Fahrenheit temperature	degree Fahrenheit	degF
Celsius temperature difference	Celsius degree	Cdeg
Fahrenheit temperature difference	Fahrenheit degree	Fdeg
length	foot	ft
length	inch	in
length	mile	mi

area	acre	acre
pressure	atmosphere	atm
energy	calorie	cal
energy	British thermal unit	Btu

Extended Precision

Double Precision

Single Precision

Long Precision

Word Precision

Byte Precision

Unsigned Long Precision

Unsigned Word Precision

Unsigned Byte Precision

Complex Extended Precision

Complex Single Precision

Complex Double Precision

Adapt to Source

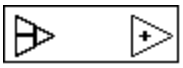
Block Diagram Introduction

This topic describes terminals and nodes--two of the three elements you use to build a block diagram (the third being wires). The topic also discusses how to get online help when you are putting these items together.

Terminals and Nodes

You create block diagrams with terminals, nodes, and wires.

Terminals are ports through which data passes between the block diagram and front panel, as well as between nodes on the block diagram. Terminals underlie the icons of functions and VIs. The following illustration shows an example of a terminal pattern on the left and its corresponding icon on the right. To display the terminals for a function or VI, pop up on the icon and select **Show Terminals**.



Nodes are program execution elements. They are analogous to statements, operators, functions, and subroutines in conventional programming languages.
Wires are the data paths between input and output terminals.

Terminals

LabVIEW has many types of terminals. In general, a terminal is any point to which you can attach a wire. LabVIEW has control and indicator terminals, node terminals, *constants*, and specialized terminals on structures.

Terminals that supply data, such as front panel control terminals, node output terminals, and constants, are also called *source terminals*. Node input terminals and front panel indicator terminals are also called *destination* or *sink terminals* because they receive the data.

Control and Indicator Terminals
Constants

Control and Indicator Terminals

You enter values into front panel controls and, when a VI executes, the control terminals pass these values to the block diagram. When the VI finishes executing, the output data passes from the block diagram to the front panel indicators through the indicator terminals.

The symbols for some of the LabVIEW control and indicator terminals are shown in the LabVIEW Control and Indicator Terminal Symbols table. Notice that each encloses a picture that suggests the data type of the control or indicator and, in the case of numerics, the representation as well. Control terminals have a thicker border than indicator terminals. Because a terminal belongs to its corresponding control or indicator, you cannot delete a terminal; if you want to delete a control, do it from the front panel.

An array terminal encloses one of the data types shown in square brackets, and takes on the color of the data type of the array.

LabVIEW Control and Indicator Terminal Symbols

Control	Indicator	Description
		Extended-precision floating-point
		Double-precision floating-point
		Single-precision floating-point
		Complex extended-precision floating-point
		Complex double-precision floating-point
		Complex single-precision floating-point
		Unsigned 32-bit integer
		Unsigned 16-bit integer
		Unsigned 8-bit integer
		32-bit integer (long word)
		16-bit integer (word)
		8-bit integer
		Cluster
		Array
		Enumeration
		Path
		Refnum

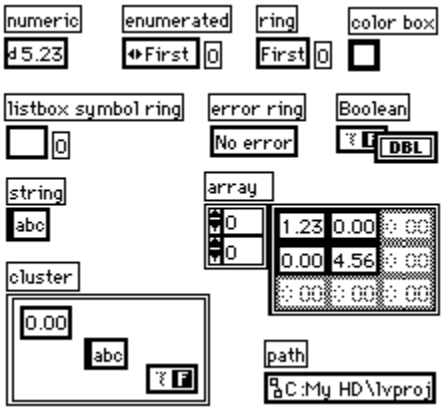
Constants

Constants are terminals on a block diagram that supply data values directly to the block diagram. User-defined constants are defined prior to program execution, and you cannot change their values during execution. Universal constants have fixed values.

User-Defined Constants

The easiest way to create a constant is to pop up on an input or output and select **Create Constant**. These constants are also available from various palettes in the **Functions** palette, depending on their type. Most are located at the bottom or top of the relevant palette, such as the numeric, enumerated, and ring constants in the bottom row or the Numeric palette. Three constants--the color box, listbox symbol ring, and error ring constants--are located in the **Numeric**»**Additional Numerics** subpalette. The path constant is in the **File I/O**»**File Constants** subpalette.

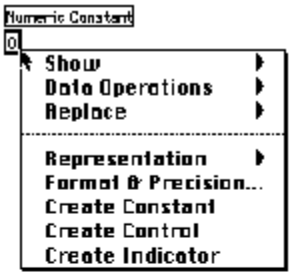
The constants are shown in the following illustration:

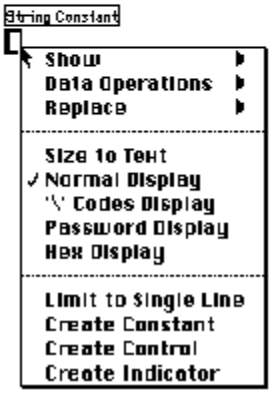


You can label your constant by popping up on the constant and selecting **Show»Label**. A highlighted text box appears, indicating that you can type into it immediately without changing the tool.

The user-defined constants, like labels, resize automatically as you enter information into them. If you resize or change the shape of a label or string constant, you can select **Size to Text** from its pop-up menu, and the constant or label resizes itself automatically to fit its contents.

You use the Operating tool to set the value of a user-defined constant the same way you set the value of a digital control, Boolean slide switch, or string control on the front panel. The numeric and string constants resemble front panel numeric controls and have pop-up menus similar to the pop-up menus of controls, as shown in the following illustration.





You can use the arrow keys to increment or decrement a new numeric constant or one with its value selected. This is particularly useful when programming control parameters with low values like 1, 2, or 3.

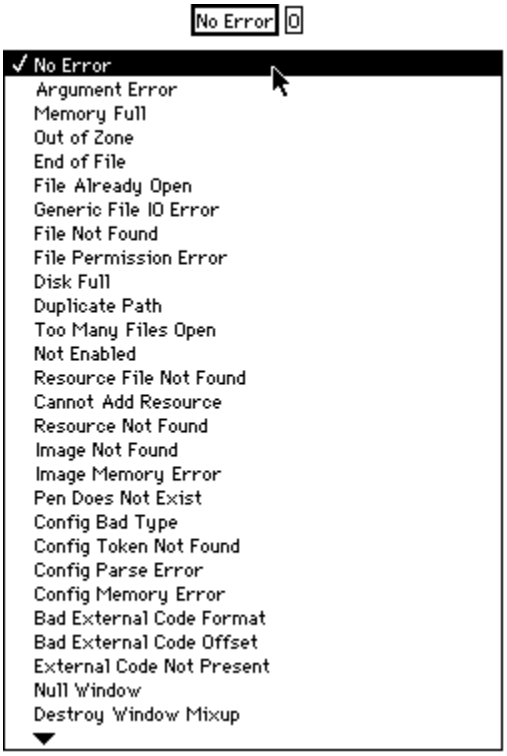
The enumerated constant is similar to the ring constant except that the mnemonics (strings associated with an integer value) are considered part of the type. When an enumeration is wired to the selection terminal of a Case Structure, cases are named according to the enumeration's mnemonics rather than traditional numeric values. The numeric representation of an enumeration is always an unsigned byte, word, or long.

The ring constant associates text with a number, the same way a ring control on the front panel does. The value of the ring constant is an unsigned 16-bit integer. Although you can change the representation of a ring constant to any numeric type except complex, the value is still always an integer from 0 to *n*-1.

You can select colors from the color box constant to use with the color box control, a control whose values correlate to specific colors. Set the color box by clicking on it with the Color tool or the Operating tool and choosing the color you want from the Color palette.

The listbox symbol ring constant is used to assign symbols to items in a listbox control.

The error ring constant is a predefined ring. You click on the constant with the Operating tool and select the error message you want from the dialog box that appears, which is shown in the following illustration. This ring is useful with the file I/O functions, because it makes the diagrams more descriptive. For example, if you try to open a nonexistent file using the Open File function, the function returns an error code of 7. You can test for this condition by comparing the error code output to an error ring set to a value of File Not Found (7).



You can use the path constant to create a constant path value on the diagram.
The path, string, and color box constants are resizable, while the Boolean, numeric, ring, and enumeration constants are not. The default representation of the numeric constant is a double-precision floating-point number if you enter a floating-point number, a long integer if you enter an integer, or a double-precision complex number if you enter a complex number. For example, the representation is long integer if you enter '123' and a double-precision floating-point number if you enter '123.'. You can change the representation with the **Representation** option from the constant pop-up menu.

Universal Constants

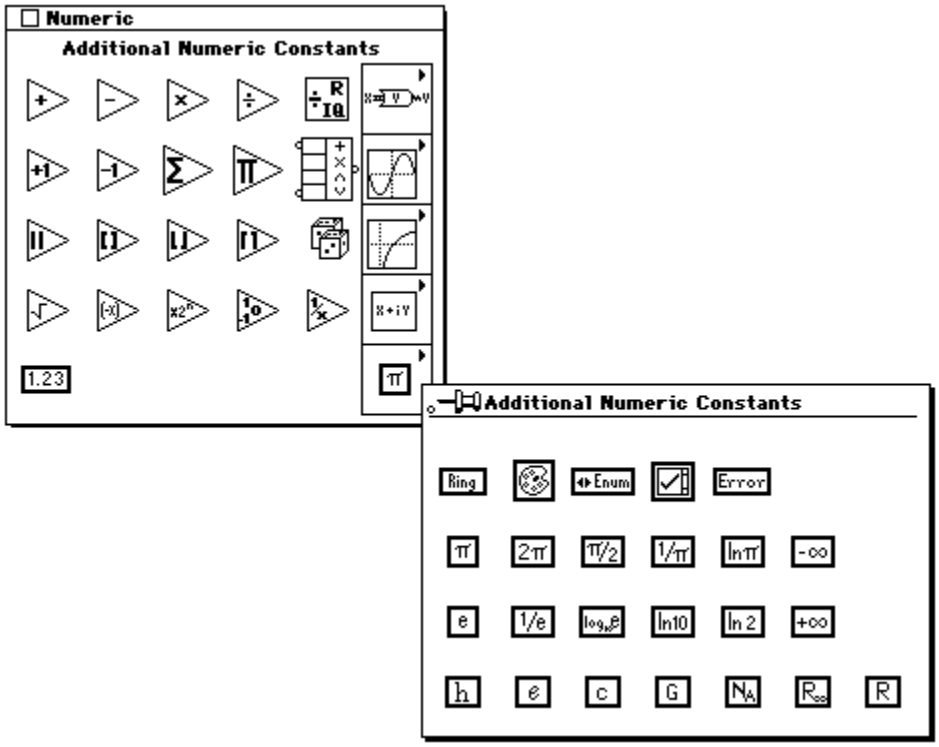
Universal constants are of two types: universal numeric constants and universal string constants.

Universal Numeric Constants

Universal numeric constants are a set of high-precision and commonly used mathematical and physical values, such as pi (p) and the speed of light (c). The Universal Numeric Constants table lists these constants, whose values reflect the precision of the extended-precision floating-point number in LabVIEW or the precision to which they are known, in the case of physical constants.

Universal Numeric Constants Table

The universal numeric constants are accessed from the **Additional Numeric Constants** at the bottom right of the **Numeric** palette.



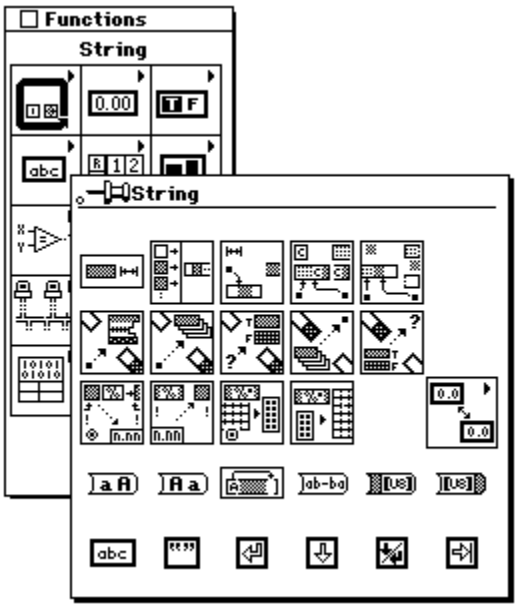
Universal String Constants

Universal string constants include five commonly used nondisplayable string characters. The Universal String Constants table shows the symbols and names for these constants.

Universal String Constants

Icon	Name
	Carriage return
	Line feed
	Tab
	Empty String
	End of Line

These constants are located in the bottom row of the **String** palette of the **Functions** palette, as shown in the following illustration.



The end of line string constant allows you to easily port your VIs from one platform to another, without needing to remember the exact characters to create a new line. In Windows, end of line is a carriage return followed by linefeed; on the Macintosh, it is a carriage return; and in UNIX, it is linefeed.

Nodes

Nodes are the execution elements of a block diagram. The six types of nodes are *functions*, *subVIs*, *structures*, *Code Interface Nodes* (CINs), *Formula Nodes*, and *Attribute Nodes*. Functions and subVI nodes have similar appearances and roles in a block diagram, but they have substantial differences.

CINs are interfaces between the block diagram and code you write in conventional programming languages such as C or Pascal.

Structures, which supplement the LabVIEW dataflow programming model to control execution order, are discussed briefly in the following Structures on the Block Diagram topic; for in-depth information, see the Structures topic.

Formula Nodes, which supplement the functions by allowing you to use formulas on the block diagram, are discussed in the Formula Node topic.

Attribute Nodes, which change control attributes programmatically, are discussed in Attribute Nodes.

Functions
Structures on the Block Diagram

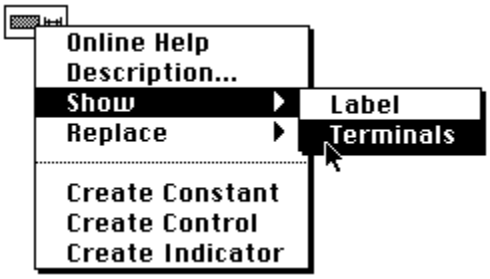
Functions

Functions are elementary nodes built into LabVIEW. They perform elementary operations like adding numbers, file I/O, and string formatting. LabVIEW functions do not have front panels or block diagrams. When compiled, they generate inline machine code.

You select functions from the **Functions** palette, as shown in the following illustration.

When you select a function, its icon appears on the block diagram. To display its label, pop up on the icon and select **Show»Label**. You can change the label if you want by highlighting the text with the Labeling tool and typing over it. You can use the function label to annotate its purpose in the diagram.

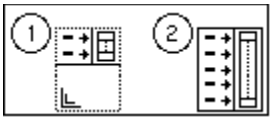
You can use the Help window to see how to wire to the function, or you can select **Show»Terminals** from the icon pop-up menu, as shown in the following illustration to see precisely where the terminals are located.



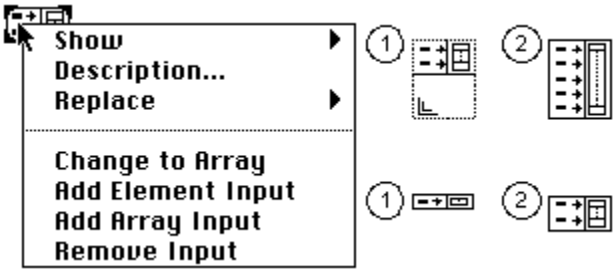
When terminals are showing, select **Show»Terminals** again to show the function's icon instead. When you wire to a function, you wire to one of its terminals.

Some array and cluster functions have a variable number of terminals. For example, if you build an array of three elements, the Build Array function needs three input terminals, but if you build one with 10 elements, the function needs 10 terminals.

You can change the number of terminals of the *expandable* functions by resizing the icon with the Resizing tool in the same way you resize other LabVIEW objects, as shown in the following illustration. You can enlarge or reduce, but you cannot shrink a function if it would cause any wired terminals to disappear.



You can also change the number of terminals with the **Add** and **Remove** commands from a terminal's pop-up menu, as shown in the illustration that follows. The **Remove** command removes the terminal on which you popped up and disconnects the wire, if the terminal is wired, while the **Add** command adds a terminal immediately after it. The full names of these commands vary with the function.



Structures on the Block Diagram

When you are programming, you sometimes need to repeat sections of code a set number of times or while a certain condition is true. In other situations, you need to execute different sections of code when different conditions exist or execute code in a specific order. LabVIEW contains four special nodes, called *structures*, that help you do these things, which otherwise are not possible within the LabVIEW dataflow framework.

Each structure has a distinctive, resizable border that you use to enclose the code that executes under the structure's special rules. For example, the diagram contained within a For Loop structure repeats execution a set number of times. For this reason, the diagram inside the structure border is called a subdiagram. You can nest subdiagrams.

Besides the For Loop, LabVIEW also has a While Loop structure, which repeats the execution of its subdiagram while a condition is TRUE; a Case Structure, which has multiple subdiagrams, only one of

which executes depending on the value passed to its selector terminal; and a Sequence Structure, which executes code in the numeric order of its subdiagrams.

Because structures are nodes, they have terminals that connect them to other nodes. For example, the terminals that feed data into and out of structures are called *tunnels*. Tunnels are relocatable connection points for wires from outside and inside the structures. See the Structures topic for more information.

Online Help for Constants, Functions, and SubVI Nodes

The LabVIEW Help window displays the wiring diagrams of functions and subVI nodes and the values of universal constants. For functions and subVI nodes, it also displays a description of the node's functionality. To display the window, choose **Show Help** from the **Help** menu, or press <Ctrl-h> (Windows); <command-h> (Macintosh); <meta-h> (Sun); or <Alt-h> (HP-UX). If your keyboard has a <Help> key, you can press that key instead.

For most block diagram objects, you can also select **Online Reference** from the object's pop-up menu to access the online description of the object. You can also access this information by pressing the button shown to the left, which is located at the bottom of LabVIEW's Help window.
For information on creating your own online reference files, see the Creating Your Own Help Files topic.
For more information about the Help window, see the Getting Help topic.

Universal Numeric Constants Table

Icon	Name	Value
π	Pi	3.14159265358979320
2π	2 Pi	6.28318530717958650
π/2	Pi Divided by 2	1.57079632679489660
1/π	Reciprocal of Pi	0.31830988618379067
lnπ	Natural Log of Pi	1.14472988584940020
-∞	Negative Infinity	- infinity
+∞	Positive Infinity	infinity
e	Natural Logarithm Base	2.71828182845904520
1/e	Reciprocal of e	0.36787944117144232
log10e	Common Logarithm of e	0.43429448190325183
ln10	Natural Logarithm of 10	2.30234095236904570
ln 2	Natural Logarithm of 2	0.69314718055994531
h	Planck's Constant (J/Hz)	6.6262e-34
e	Elementary Charge (C)	1.6021892e-19
c	Speed of Light (m/sec)	299,792,458
G	Gravitational Constant (N m2/kg2)	6.6720e-11
N_A	Avogadro Number (1/mol)	6.0220e23
R_∞	Rydberg Constant (/m)	1.097373177e7
R	Molar Gas Constant (J/mol K)	8.31441

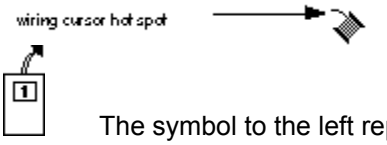
Wiring the Block Diagram

This topic explains how to connect terminals on the block diagram by wiring.

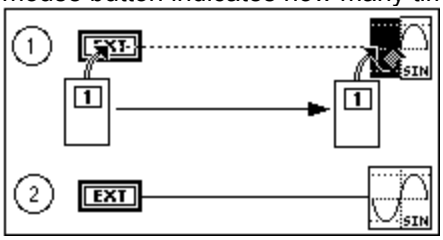
Wiring Techniques
Replacing and Inserting Block Diagram Objects
Adding Constants, Controls, and Indicators Automatically
Common Reasons for Bad Wires
Wiring Situations to Avoid
Problems in Wiring Structures

Wiring Techniques

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



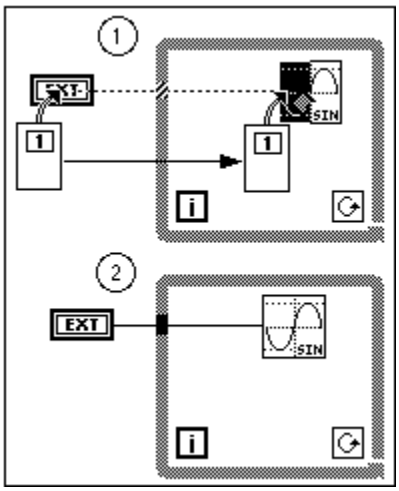
The symbol to the left represents the mouse. In subsequent topics, the wiring illustrations contain an arrow at the end of this mouse symbol, which shows where to click, and the number printed on the mouse button indicates how many times to click.



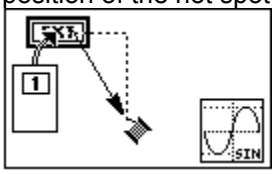
To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and then click on the second terminal as shown in the preceding illustration. It does not matter which terminal you click on first. The terminal area blinks when the hot spot of the Wiring tool is correctly positioned on the terminal. Clicking connects a wire to that terminal. Once you have made the first connection, LabVIEW draws a wire as you move the cursor across the diagram, as if the wire were reeling off the spool. You do not need to hold down the mouse button.

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

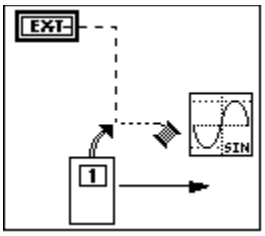
You can wire directly from a terminal outside a structure to a terminal within the structure using the basic wiring operation. LabVIEW creates a *tunnel* where the wire crosses the structure boundary, as shown in the following illustration.



Wires reel off from terminals vertically or horizontally, depending on the direction in which you first move the Wiring tool. Wires spool vertically if you move the tool up or down, and they spool horizontally if you move the tool left or right. LabVIEW centers the connection on the terminals, regardless of the exact position of the hot spot when you click the mouse button as shown in the following illustration.

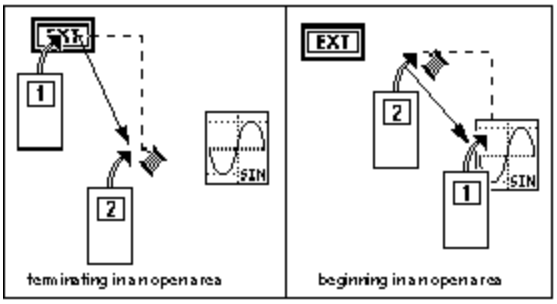


You can elbow your wire once without clicking. You click the mouse to tack the wire and change direction, as shown in the following illustration.

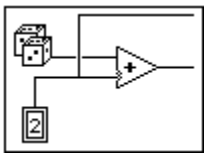


Note: You can also change between horizontal and vertical direction by pressing the spacebar. You can untack the last tack point by pressing <`Ctrl-click`> (Windows); (<`option-click`> (Macintosh); <`meta-click`> (Sun); <`Alt-click`> (HP-UX). If the last tack point is the terminal or wire upon which you first clicked, untacking removes the wire completely.

You can double-click with the Wiring tool to begin or terminate a wire in an open area as shown following.



When wires cross, a small gap appears in the first wire drawn, as if it were underneath the second wire, as shown here.

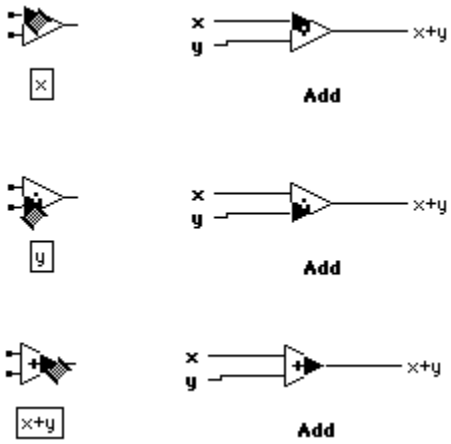


Wiring Complicated VIs
Wire Stretching
Wire Selecting, Moving, and Deleting
Wiring to Off-Screen Areas
Duplicating Sections of the Block Diagram

Wiring Complicated VIs

When you are wiring a complicated built-in node or subVI, it helps to pay attention to the wire stubs and the tip-strips that appear as the Wiring tool approaches the VI icon. Wire stubs, the truncated wires shown around the VI icon to the left, indicate the data type by their style, thickness, and color. (For details, see the “LabVIEW Quick Reference Card.”) Dots at the end of the stubs indicate inputs, while outputs have no dots. The direction in which the stubs are drawn indicates the suggested direction to wire to produce clean diagrams.
When a terminal is wired, the wire stub for that terminal is no longer displayed.

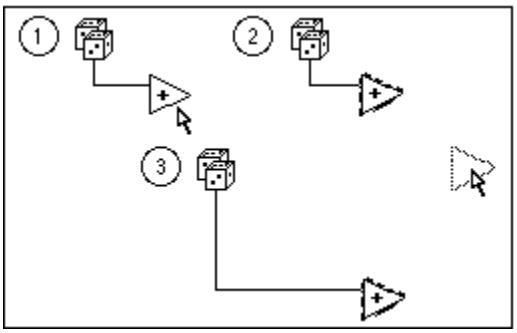
You may also want to take advantage of the Help window feature that highlights each connector pane terminal. With this feature you can see exactly where wires need to connect. The three connections of the Add function are shown as a simple example in the following illustration.



The feature just illustrated does not work for growable functions, like the Array Bundle function for example.

Wire Stretching

You can move wired objects individually or in groups by dragging the selected objects to the new location using the Positioning tool. The wires connected to the selected objects stretch automatically. The wire stretching capability is demonstrated in the following illustration.

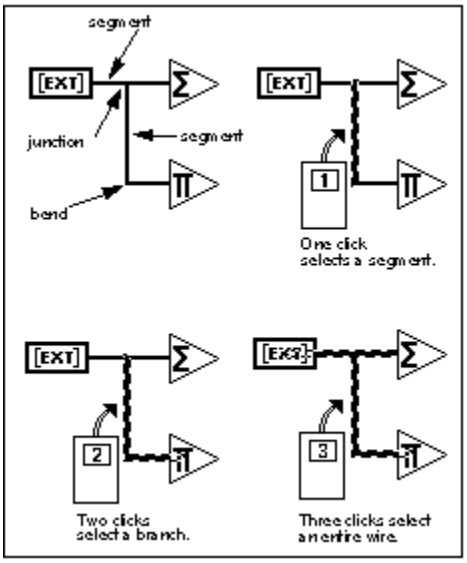


If you duplicate the selected objects or move them from one diagram into another--for example, from the block diagram into a structure diagram--LabVIEW leaves behind the connecting wires, unless you select them as well.

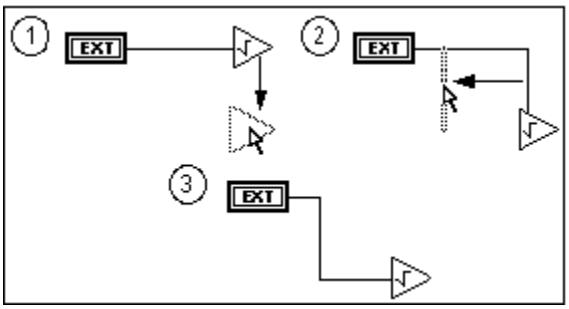
Note: Wire stretching occasionally creates wire stubs or loose ends. You must remove the stubs or loose wires before the VI will execute. The easiest way to do this is to select the Edit»Remove Bad Wires command.

Wire Selecting, Moving, and Deleting

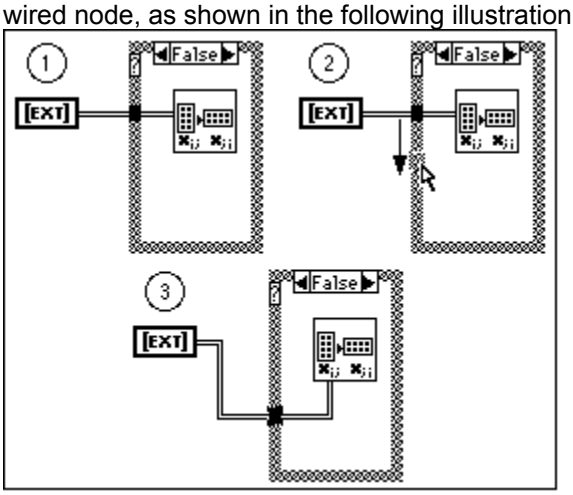
A wire *segment* is a single horizontal or vertical piece of wire. The point at which three or four wire segments join is a *junction*. A *bend* in a wire is where two segments join. A wire *branch* contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal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



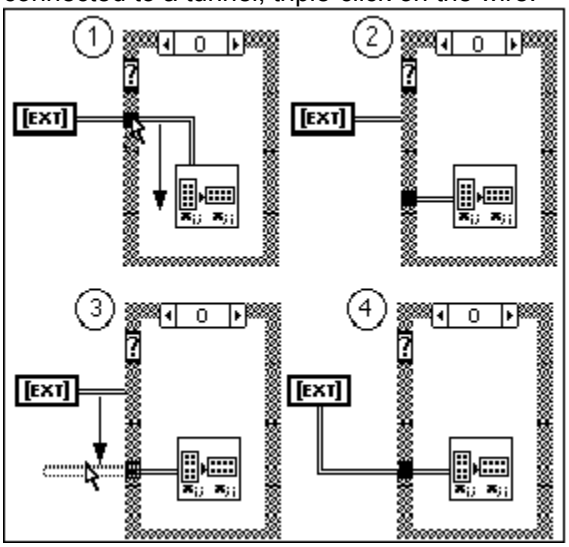
To reposition a wire segment, drag it to the new location with the Positioning tool. You can reposition one or more segments by selecting and dragging them. You can also move selected segments one pixel at a time by pressing the arrow keys on the keyboard. LabVIEW stretches adjacent, unselected segments to accommodate the change. You can select and drag multiple wire segments, even discontinuous segments, simultaneously as shown in the following illustration.



Sometimes repositioning a wired object results in an extra segment or oddly positioned wire segment, as shown in the preceding example. If you do not like the way the wire looks, you can move it. You can use the <Shift> key to restrict the wire drag in horizontal or vertical direction. The direction in which you initially move determines whether the wire is limited to horizontal or vertical translation.
When you move a tunnel, LabVIEW normally maintains a wire connection between the tunnel and the wired node, as shown in the following illustration.



Moving a tunnel sometimes creates an extra wire segment that lies beneath the structure border, however. You cannot select and drag this segment because it is hidden, but it disappears if you drag the segment connected to it, as shown in the following illustration. If you are unsure about which wire is connected to a tunnel, triple-click on the wire.



To select the parts of a wire inside and outside a loop structure at the same time, select the part of the wire on one side of the structure and hold down the <Shift> key while you select the part of the wire on the other side of the structure. You can add an object to a group of previously selected objects by holding

down the <Shift> key while you select the new object. You can also drag a selection rectangle around both parts of the wire. (The structure will not be selected unless you completely surround it with a selection rectangle. Other nodes need only touch the rectangle to be selected.)

Wiring to Off-Screen Areas

If a block diagram is too large to fit on the screen, you can use the scroll bars to move to an off-screen area and drag whatever objects you need there.

You can automatically scroll the diagram while you are wiring by dragging the Wiring tool slightly past the edge of the block diagram window.

Duplicating Sections of the Block Diagram

As with front panel editing, you can use the Clipboard to copy, cut, and paste objects from the block diagram of one VI to another. When you duplicate the front panel control and indicator terminals, you will also duplicate the corresponding front panel controls and indicators on the front panel of the destination VI.

For example, if you need to use a portion of a block diagram in another VI, you can select that portion with the Positioning tool, copy it to the Clipboard, and paste it into the new VI. You can also drag the selected portion of the diagram directly to the other VI's diagram. When you copy portions of a diagram between VIs, consider turning that portion into a subVI with an icon/connector.

Replacing and Inserting Block Diagram Objects

Suppose you used an Increment function in the block diagram where you should have used the Decrement function. You can delete the Increment function node and then select the Decrement node from the **Functions** palette and rewire. You can also use the **Replace** option in the Increment node pop-up menu. Selecting **Replace** gives you the **Functions** palette, from which you can choose the Decrement function. The advantage of this method is that LabVIEW places the new node where the old node was and does not disturb the wiring. You can replace a function with any other function, although if the number of terminals or data types in each function node is different, you may get broken wires.

You can also use **Replace** to replace a constant with another constant or a structure with another similar structure, such as a While Loop with a For Loop.

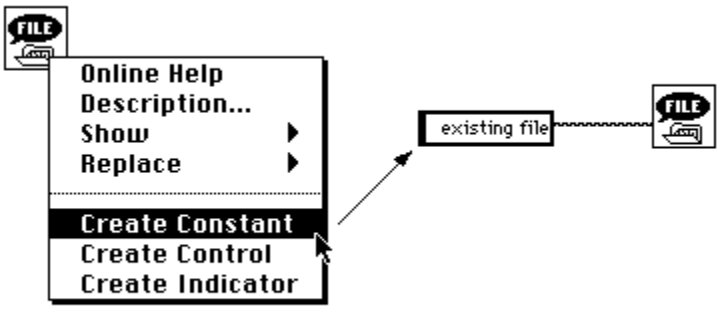
Note: When using Replace, if you select a VI whose name is the same as one already in memory, the replaced node will refer to the VI that was already in memory, not the VI you selected.

Wire pop-up menus have an **Insert** option. Choosing **Insert** accesses the **Functions** palette, from which you can choose any function or VI on the menu. LabVIEW then splices the node you choose into the wire on which you popped up. You must be careful to check the wiring if the node has more than one input or output terminal, however, because the wires may not connect to the terminal you expected.

Adding Constants, Controls, and Indicators Automatically

Instead of creating a constant, control, or indicator by selecting it from a menu and then wiring it manually to a terminal, you can pop up on the terminal and choose **Create Constant, Create Control**, or **Create Indicator** to create an object with an appropriate data type automatically. Assuming it makes sense to do so, the constant, control, or indicator created is automatically wired for you.

For example, if you need a constant for the position mode input of a File I/O function, you can pop up on the input and select **Create Constant**. An enumerated type is created for you. Assuming the mode input isn't already wired to something else, the new constant is automatically wired.



Other useful places to pop up include the outputs of functions or VIs, constants, and terminals for front panel controls or indicators.

Common Reasons for Bad Wires

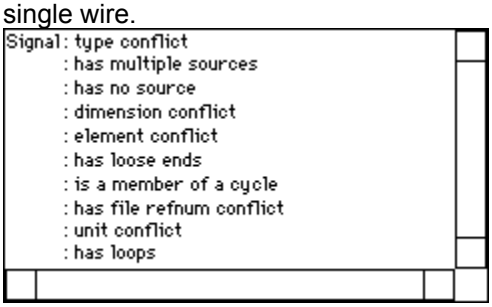
Faulty connections describe some of the common causes of faulty wires.

Faulty Connections

If a wire connection is faulty, the wire is broken. If you already know why a wire is broken, choose Edit»**Remove Bad Wires** and rewire the affected objects correctly. Sometimes you may have a faulty wiring connection that is not visible because the offending segment is very small or is hidden behind an object. If the run button shows a broken arrow  , but you cannot see any problems in the block diagram, select **Remove Bad Wires** in case there are hidden bad wire segments. If the run button returns to its unbroken state

 , you corrected the problem. If not, click on the broken run button to see a list of errors.
If you do not know why a particular wire is broken, pop up on the broken wire and choose **List Errors** from the pop-up menu. The dialog box shown in the following illustration appears and lists the errors. Click on one of the errors. Doing this selects the erroneous wire, which you can then delete or repair.

For more information, see the Debugging VIs topic.
The following figure shows a list of some possible wire errors. They cannot all occur simultaneously for a single wire.

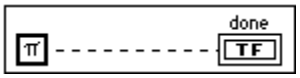


Wire Type, Dimension, Unit, or Element Conflict
Multiple Wire Sources
No Wire Source
Loose Ends
Wire Stubs

Wire Cycle

Wire Type, Dimension, Unit, or Element Conflict

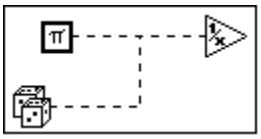
A type mismatch occurs when you wire two objects of different data types together, such as a numeric and a Boolean as shown in the following illustration.



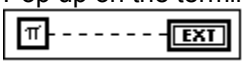
A file refnum conflict results when you wire refnums for two datalog files of different record types.
The dimension conflict and element conflict errors occur in similar situations--when you wire two arrays together whose elements match but whose dimensions do not, and when you wire two clusters whose elements have type differences, respectively. The unit conflict occurs when you wire together two objects that do not have commensurable units.
These problems typically arise when you inadvertently connect a wire to the wrong terminal of a function or subVI. Select and remove the wire and rewire to the correct terminal. You can use the Help window to avoid this type of error. In other situations, you may have to change the type of one of the terminals.

Multiple Wire Sources

You can wire a single data source to multiple destinations, but you cannot wire multiple data sources to a single destination. In the example that follows you must disconnect one source.

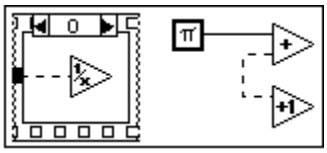


During front panel construction, you may have dropped a control when you meant to drop an indicator. If you try to wire an output value to the terminal of a front panel control, you get a multiple sources error. Pop up on the terminal and select the **Change To Indicator** command to correct this error.



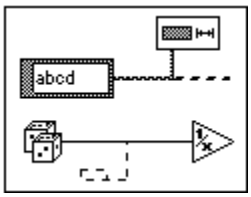
No Wire Source

Two examples of wires with no sources are shown in the following illustration. In one case, a tunnel supplies data to the Reciprocal function, but nothing supplies data to the tunnel. In the other case, two function inputs are wired together, but there is no data source for them. The solution to the problems is to wire a data source to the tunnel and to the Add and Increment functions, respectively. You also get this error if you wire together two front panel indicator terminals when one should be a control. In this case, pop up on one terminal and select the **Change To Control** command.



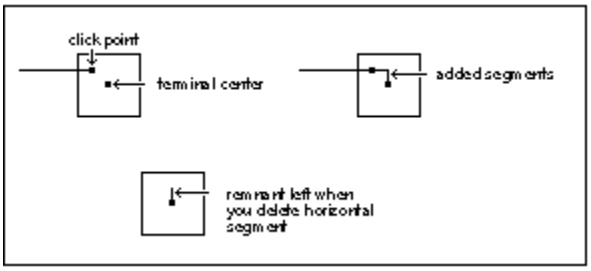
Loose Ends

Loose ends, shown in the illustration that follows, are branches of wire that do not connect to a terminal. These can result from wire stretching, from retracing during the wiring process, or from deleting wired objects. Selecting **Remove Bad Wires** disposes of loose ends.



Wire Stubs

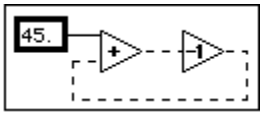
When you wire to a terminal, you seldom click when the cursor is exactly at the center of the terminal. When you are off center, *LabVIEW automatically adds a wire segment from the click point to the terminal center*. If you then remove the wire going to the terminal, a wire stub can remain, causing a broken run button. This problem is shown in the following illustration.



To avoid this situation, triple click on a wire to make sure all portions are selected, then delete it. You can also use the **Remove Bad Wires** option in the **Edit** menu to delete the stubs.

Wire Cycle

Wires must not form cycles. That is, wires must not form closed loops of icons or structures as shown in the following illustration. LabVIEW cannot execute cycles because each node waits on the other to supply it data before it executes.



The Shift Registers topic describes the correct way to feed back data in a repetitive calculation.

Wiring Situations to Avoid

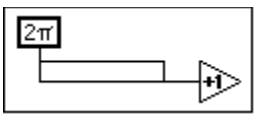
The following topics describe situations that do not produce bad wires but do make the block diagram difficult to read or make it appear to do things it actually does not do.

Note: Remember, when you are unsure of what connects to a wire you can double-click or triple-click on the wire to select the branch or the entire wire.

Wire Loops
Hidden Wire Segments
Wiring Underneath Objects

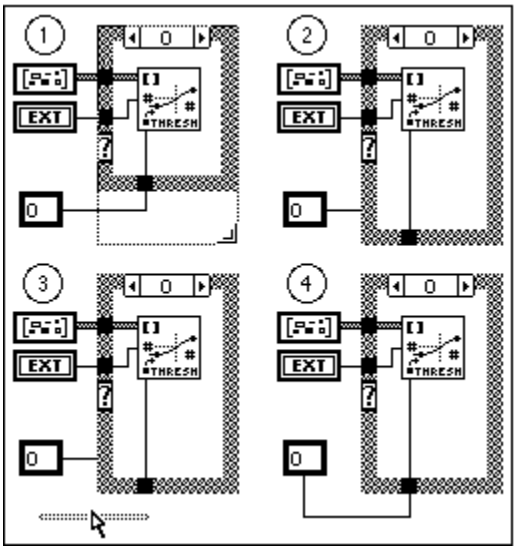
Wire Loops

A loop of wire is not an error but is poor design because it unnecessarily clutters the diagram. Double-click on one of the branches to select it, then delete it. A wire loop is shown in the following illustration.



Hidden Wire Segments

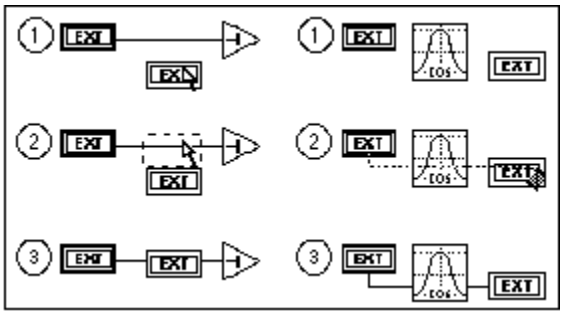
Try to avoid wiring under a structure border or between overlapped objects, because some segments of the resulting wire may be hidden. An example of hidden wire segments is shown in the following illustration.



You can inadvertently create hidden wire segments, such as when you move a tunnel or enlarge a structure, as shown in parts 1 and 2 of the preceding example. Parts 3 and 4 of this example show one way you can make this diagram less confusing. You can drag the wire segment connected to the constant so that the hidden wire segments reappear. Press the <Shift>key to constrain the wire drag and reduce the likelihood of creating loose ends.

Wiring Underneath Objects

Wires connect only those objects that you click on. Dragging a terminal or icon on top of a wire makes it appear as if a connection exists when it does not, as shown at the left of the illustration that follows (1, 2, 3 on the left).



Dragging a wire through an icon or terminal also appears to make a connection, but the wire is actually behind the icon, as shown at the right of the illustration (1, 2, 3 on the right). Avoid these situations because they are visually confusing.
See Warnings for more information on this problem.

Problems in Wiring Structures

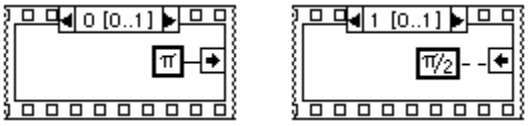
The following topics discuss faulty connections with structures.

Assigning More Than One Value to a Sequence Local
Failing to Wire a Tunnel in All Cases of a Case Structure
Overlapping Tunnels

Wiring from Multiple Frames of a Sequence Structure
Wiring Underneath Rather Than through a Structure
Removing Structures without Deleting Contents

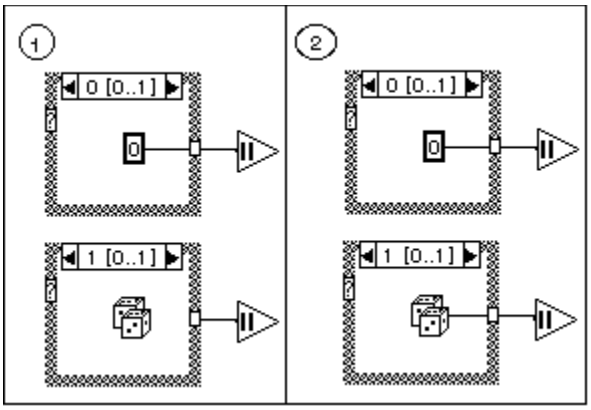
Assigning More Than One Value to a Sequence Local

You can assign a value to the local variable of a Sequence Structure in only one frame, although you can use the value in all subsequent frames. The illustration to the left below shows the value pi assigned to the sequence local in frame 0. If you try to assign another value to this same local variable in frame 1, you get a bad wire. This error is a variation of the multiple sources error.



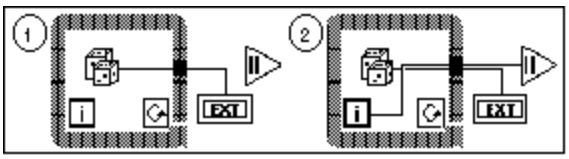
Failing to Wire a Tunnel in All Cases of a Case Structure

Wiring from a Case Structure to an object outside the structure results in a bad tunnel if you do not connect a source in all cases to the object, as shown in part 1 of the following example. This is a variation of the no source error because at least one case would not provide a data value if it executed. Wiring to the tunnel in all cases, as shown in part 2 of this example, corrects the problem. This is not a multiple sources violation because only one case executes and produces only one output value per execution of the Case Structure.

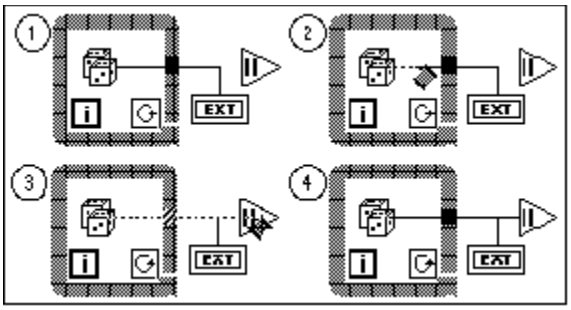


Overlapping Tunnels

Because LabVIEW creates tunnels as you wire, tunnels sometimes overlap. Overlapping tunnels do not affect the execution of the diagram, but they can make editing difficult. You should avoid creating overlapping tunnels. If they occur, drag one tunnel away to expose the other. Look at the following example.



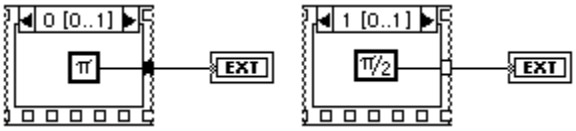
It is difficult to tell which tunnel is on top. You can make mistakes if you try to wire to one of them while they overlap, although you can always remove the bad wires and try again.



If you need to wire from an object inside a structure to an object outside when one such wire already exists, *do not wire* through the structure again as shown in the above illustration. Instead, begin the second wire at the tunnel. In this example, two overlapping tunnels do not cause a problem. But if this were a Case Structure, two overlapping bad tunnels might have appeared to be wired in each case. You can always remove all the wires from a tunnel to make it vanish and then rewire correctly.
If your VI has tunnels that are completely overlapping, a warning appears in the Error List window if you have selected **Show Warnings**. See Warnings for more information on these problems.

Wiring from Multiple Frames of a Sequence Structure

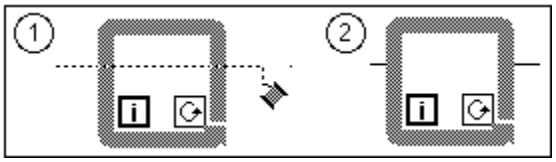
This next illustration shows another variation of the multiple sources error. Two Sequence Structure frames attempt to assign values to the same tunnel. The tunnel turns white to signal this error.



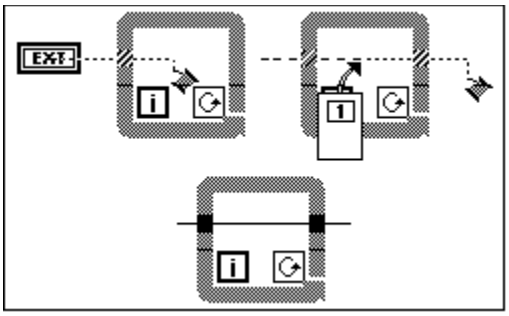
Wiring Underneath Rather Than through a Structure

To wire through a structure you must click either in the interior or on the border of the structure, as shown below.

If you do not click in the interior or on the border of the structure, the wire passes underneath the structure, as shown below.



When the Wiring tool crosses the left border of the structure, a highlighted tunnel appears to indicate that LabVIEW will create a tunnel at that location as soon as you click the mouse button. If you continue to drag the tool through the structure without clicking the mouse until the tool touches the right border of the structure, a second highlighted tunnel appears on the right border. If you continue to drag the Wiring tool past the right border of the structure without clicking, both tunnels disappear, and the wire passes underneath the structure rather than through it.
Examine the following illustration.



If you tack down the wire inside the structure, however, the wire goes through the structure even if you continue dragging the Wiring tool past the right border.

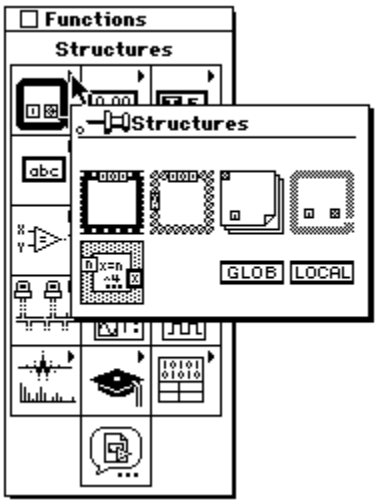
Removing Structures without Deleting Contents

You can remove a structure (While Loop, For Loop, Sequence, or Case Structure) without losing the contents of the structure. If you pop up on any of these objects, you see an option for deleting the structure. In the case of While Loops and For Loops, the contents of the loop are copied to the underlying diagram. In addition, any wires that were connected by tunnels are automatically connected together.

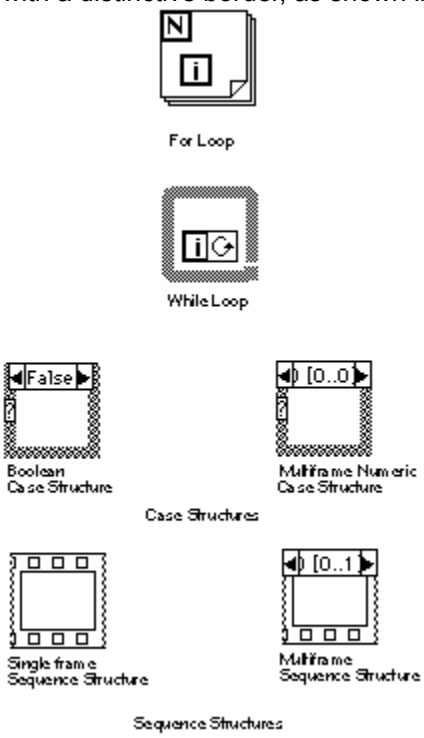
In the case of a Sequence or Case Structure, removing the structure only preserves the current frame or case. All other frames or cases will be deleted, and you are warned that you will lose hidden frames or cases and given a chance to cancel the operation.

Structures

This topic describes how to use the For Loop, While Loop, Case Structure, and Sequence Structure. These structures are in the **Structures** palette, accessed from the **Functions** palette, as shown in the following illustration:



See `examples\general\structs.llb` for examples of how these structures are used in LabVIEW. Structures are nodes that supplement the flow of execution in a block diagram, just as control structures do in a conventional programming language. The icon for each LabVIEW structure is a resizeable box with a distinctive border, as shown in the following illustration.



Structures behave like other nodes in that they execute automatically when their input data is available, and they supply data to their output wires only when execution completes. However, each structure executes its subdiagram according to the rules described in the following sections.

A subdiagram is the collection of nodes, wires, terminals, and space that resides within the structure border. The For Loop and While Loop each have one subdiagram. The Case and Sequence structures,

however, can have multiple subdiagrams stacked like cards in a deck with only one visible at a time. You construct subdiagrams the same way that you construct the top-level block diagram; subdiagrams can contain block diagram terminals, nodes (including other structures), and wires.

LabVIEW creates terminals for passing data into and out of a structure automatically where wires connecting outside nodes and inside nodes cross the structure boundary. These boundary terminals are called tunnels. Tunnels always have one edge exposed to the inside of the structure and one edge exposed to the outside. A tunnel always resides on the border of the structure, but you can move it anywhere along that border by dragging it with the Positioning tool. You can think of tunnels as way stations for data flowing into or out of a structure. Depending upon the type of structure, the data may be transformed by the tunnels.

Structures also have other terminals that are particular to each type of structure.

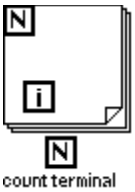
For Loop and While Loop Structures
Case and Sequence Structures

For Loop and While Loop Structures

You use the For Loop and While Loop to control repetitive operations, either until a specified number of iterations has completed (For Loop) or until a specified condition is no longer true (While Loop).

For Loop
While Loop
Placing Objects inside Structures
Placing and Sizing Structures on the Block Diagram
Terminals inside Loops
Auto-Indexing
Executing a For Loop Zero Times
Shift Registers

For Loop



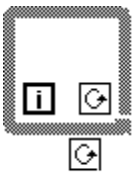
count terminal A For Loop executes its subdiagram *count* times, where the count equals the value contained in the *count terminal*. You can set the count explicitly by wiring a value from outside the loop to the left or top side of the count terminal, or you can set the count implicitly with auto-indexing. The other edges of the count terminal are exposed to the inside of the loop so that you can access the count internally.

iteration terminal The *iteration terminal* contains the current number of completed iterations; 0 during the first iteration, 1 during the second, and so on up to *N-1*. *Both the count and iteration terminals are signed long integers with a range of 0 through 2*31-1. If you wire a floating-point number to the count terminal, LabVIEW rounds it, if necessary, and coerces it to within range. If you wire 0 to the count terminal, the loop does not execute.

The For Loop is equivalent to the following pseudocode:

```
for i = 0 to N-1
   Execute subdiagram
```

While Loop



conditional terminal A While Loop executes its subdiagram until a Boolean value you wire to the *conditional terminal* is FALSE. LabVIEW checks the conditional terminal value at the end of each iteration, and if the value is TRUE, another iteration occurs, so the loop always executes at least once. The default value of the conditional terminal is FALSE, so if it is unwired, the loop iterates only once.

iteration terminal The iteration terminal behaves exactly as it does in the For Loop.

The While Loop is equivalent to the following pseudocode:

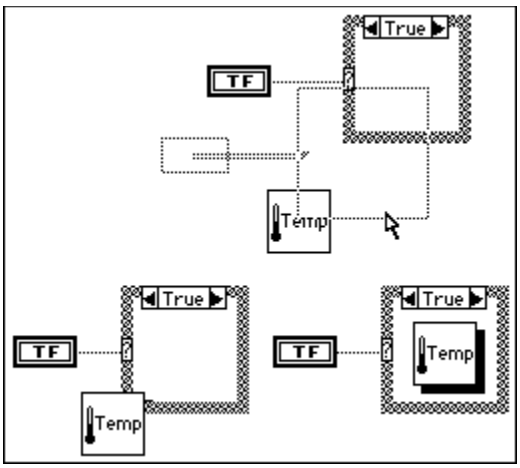
```
Do
   Execute subdiagram (which sets condition)
While condition is TRUE
```

Both loop structures can have terminals called *shift registers* that you use for passing data from the current iteration to the next iteration. See the Shift Registers topic for more information.

Placing Objects inside Structures

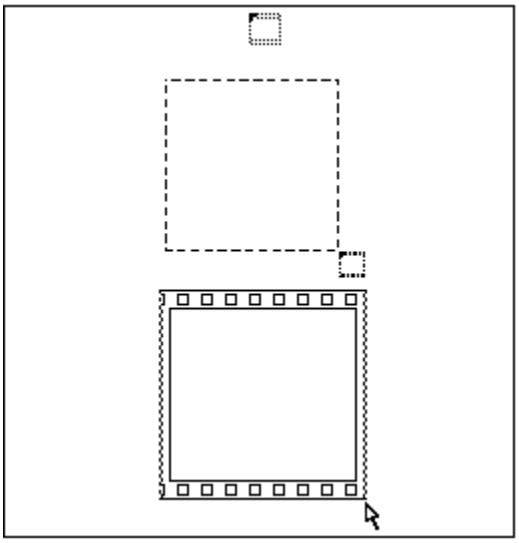
You can place an object inside structures by dragging it inside, or by creating the structure around the object.

You cannot put an object inside a structure by dragging the structure over the object. If you move a structure and it overlaps another object, LabVIEW will display whichever object it considers to be in front. If you put the structure completely over another object, and LabVIEW considers that object in front, it will display a thick shadow to warn you that the object is below, not inside the structure. Otherwise, the structure will completely hide the object. (For information on which objects LabVIEW considers frontmost, see the Moving Objects to Front and Back topic.Both situations are shown in the following illustration.



Placing and Sizing Structures on the Block Diagram

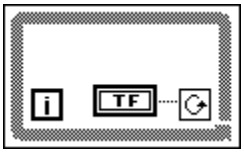
When you select a structure from the **Structures** palette and get ready to put it on your diagram, you see a small icon of the structure in place of your cursor. This icon is ready to be put on your diagram. You can either click and release your mouse button to drop a small, default-size structure on the diagram, or you can click on the diagram while holding down the mouse and drag the structure to the size you want. An example is shown in the following illustration. You can also size the structure *after* you have dropped it on the diagram by clicking on any corner and dragging.



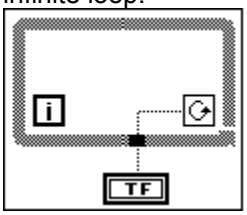
Terminals inside Loops

Inputs to a loop pass data before loop execution. Outputs pass data out of a loop only after the loop completes all iterations.

You must place a terminal *inside* a loop when you want the loop to check the terminal's value on each iteration. For example, when you place the terminal of a front panel Boolean control inside a While Loop and wire the terminal to the loop conditional terminal of the loop, the loop checks the value of the terminal at the end of *every* iteration to determine whether it should iterate again. You can stop this While Loop, shown in the following figure, by changing the value of the front panel control to FALSE.



If you place the terminal of the Boolean control outside the While Loop, as shown below, you create an infinite loop.

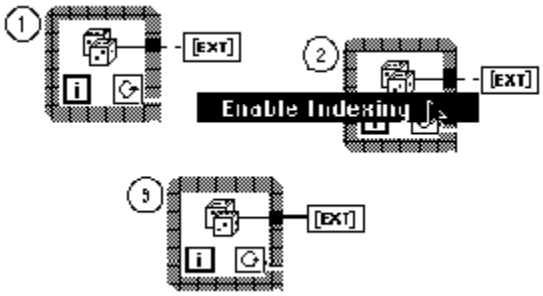


If the control was TRUE at the start, changing the value of the front panel control to FALSE does not stop the execution of this loop because the value is not propagated until the loop stops and the VI is re-run. If you inadvertently create an infinite loop in LabVIEW, you can always stop it by aborting the VI. Click on the Stop button to abort.

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 wire an array of any dimension from an external node to an input tunnel on the loop border and enable auto-indexing on the input tunnel, components of that array enter the loop one at a time, starting with the first component. The loop indexes scalar elements from one-dimensional arrays, one-dimensional arrays from two-dimensional arrays, and so on. The opposite action occurs at output tunnels–elements accumulate sequentially into one-dimensional arrays, one-dimensional arrays accumulate into two-dimensional arrays, and so on. The

following illustration shows the appearance of tunnels on the loop structure borders with and without auto-indexing. The wire becomes thicker as it changes dimensions at the loop border.



You often use For Loops to process arrays sequentially. For this reason, LabVIEW enables auto-indexing by default when you wire an array into or out of a For Loop. While Loops are not commonly used for that purpose, so LabVIEW does not enable auto-indexing for them by default. To enable or disable auto-indexing on a tunnel on the loop border, you must pop up on the tunnel at the loop border and choose **Enable Indexing** or **Disable Indexing**.

Auto-Indexing to Set the For Loop Count
Auto-Indexing with While Loops

Auto-Indexing to Set the For Loop Count

When you enable auto-indexing on an array entering a For Loop, LabVIEW automatically sets the count to the size of the array, thus eliminating the need for you to wire to the count terminal explicitly. If you enable auto-indexing for more than one tunnel, or if you do set the count explicitly, the count becomes the smallest of the choices. So if two auto-indexed arrays enter the loop, with 10 and 20 components respectively, and if you wire a value of 15 to the count terminal, the count is 10, and the loop indexes only the first 10 components of the second array.

Auto-indexing output arrays receive a new output element from every iteration of the loop. Therefore, auto-indexed output arrays are always equal in size to the number of iterations (10 in the previous example).

If Auto-indexing is disabled, only the value from the last iteration of the loop is passed.

Auto-Indexing with While Loops

When you enable auto-indexing for an array entering a While Loop, the While Loop indexes the array the same way as a For Loop does. However, the number of iterations a While Loop executes is not limited by the size of the array, because the While Loop iterates as long as a certain condition is TRUE. When a While Loop indexes past the end of the array, the default value for the array element type passes into the loop. Auto-indexing output arrays receive an output element from each iteration of the While Loop. They continue to grow in size as long as the While Loop executes.

Note: Auto-indexing with a While Loop that iterates too many times can cause you to run out of memory. See Chapter 4, *Arrays and Graphs*, of the *LabVIEW Tutorial Manual* for further information about building arrays with While Loops to prevent this problem.

Executing a For Loop Zero Times

When you set the count to zero, a For Loop does not execute its subdiagram at all. The value of all scalar data leaving the For Loop conforms to the following rules.

- An output array created by auto-indexing at an output tunnel is empty.
- The output from an initialized shift register is the initial value. See the Shift Registers topic for more

information.

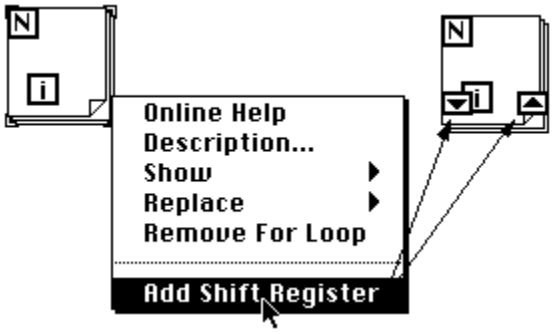
- An array passed through a non-indexing output tunnel is also empty.
- All scalars passed through non-indexing output tunnels are undefined, and you cannot rely on their value.

The loop count is set to zero or defaults to zero in two ways. You can either auto-index an empty input array, or you can wire a zero or a negative number to the count terminal explicitly. You must wire to the count terminal unless you auto-index an input array.

Whenever you auto-index input arrays or set the loop count with a variable, you should analyze the diagram to determine whether a zero count can occur, and if so, what the effects would be.

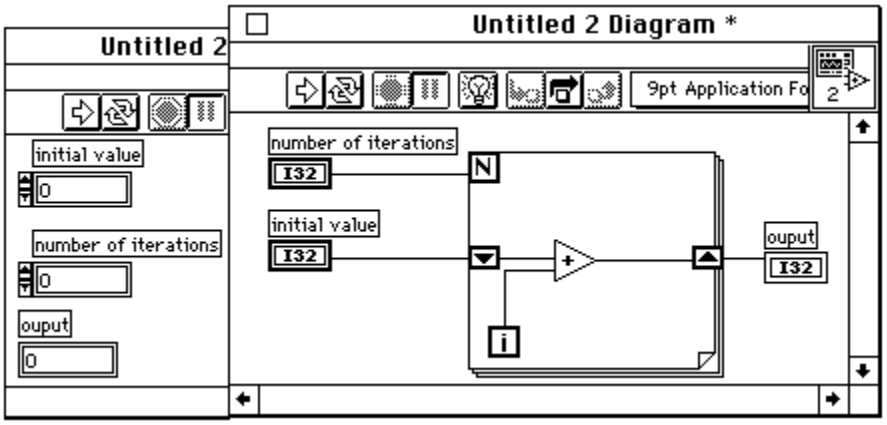
Shift Registers

Shift registers, which are available in For Loops and While Loops, are local variables that feed back or transfer values from the completion of one iteration to the beginning of the next. By selecting **Add Shift Register** from the loop border pop-up menu, you can create a register anywhere along the vertical boundary, as shown in the following illustration. This menu item is not available from the top or bottom edge of the structure. You can reposition a shift register along the vertical boundary by dragging it.

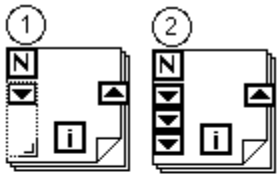


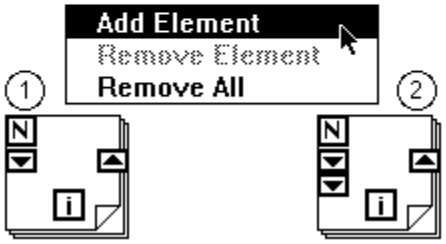
A shift register has a pair of terminals directly opposite each other on the vertical sides of the loop border. The *right terminal*, the rectangle with the up arrow, stores the data at the completion of an iteration. LabVIEW shifts that data at the end of the iteration, and it appears in the *left terminal*, the rectangle with the down arrow, in time for the next iteration. You can use shift registers for any type of data, but the data you wire to each register's terminals must be of the same type. You can create multiple shift registers on a particular structure.
To initialize a shift register, wire a value from outside the loop to the left terminal. If you do not initialize the register, the loop uses as the initial value the last value inserted in the register when the loop last executed, or the default value for its data type if the loop has never before executed. You should normally use initialized shift registers to ensure consistent behavior. See Chapter 3, *Loops and Charts*, of the *LabVIEW Tutorial Manual* for further information on using uninitialized shift registers.

When the loop finishes executing, the last value stored in the shift register remains at the right terminal. If the right terminal is wired outside of the loop, this last value passes out when the loop completes. This property is shown in the following illustration.



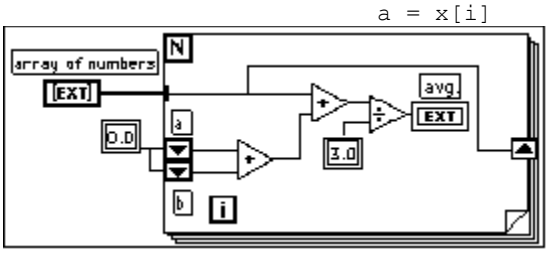
To add or remove terminals to a particular shift register, use the Positioning tool to resize the left terminals. Alternatively, you can use the **Add Element** command from the shift register pop-up menu to add more left terminals to the shift register. Added terminals appear directly below the one on which you pop up. Use the **Remove Element** command to remove the terminal on which you pop up. The following illustration shows both methods. The only way to remove extra wired terminals is to choose **Remove Element** from the shift register pop-up menu. Selecting **Remove All** deletes the shift register.





The left, topmost terminal holds the value from the previous iteration, *i*-1. The terminal immediately under the uppermost terminal contains the value from iteration *i*-2, and so on with each successive terminal.
If you initialize one left terminal of a shift register, you must initialize all of them.
The following pseudocode shows a three-value running average routine equivalent to the LabVIEW block diagram.

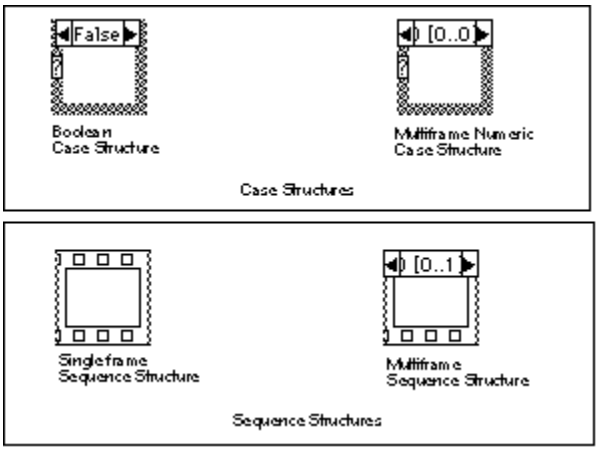
```
a = b = 0
for i = 0 to N-1
   avg = (x[i]+a+b)/3
   b = a
   a = x[i]
```



For an example of how a shift register is set up, see `examples\general\structs.llb\Random Average.vi`.

Case and Sequence Structures

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 is currently being displayed. If the diagram identifier is numeric, it is followed by a diagram identifier range, which shows the minimum and maximum values for which the structure contains a subdiagram.



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. Other uses of the display window are explained in the Editing Case and Sequence Structures topic.

See `examples\general\structs.llb\SquareRoot.vi` for an example of a Case Structure.

Case Structure
Sequence Structure
Editing Case and Sequence Structures
Adding Subdiagrams
Moving between Subdiagrams
Deleting Subdiagrams
Reordering Subdiagrams

Case Structure

The Case Structure has one or more subdiagrams, or *cases*, exactly one of which executes when the structure executes. This depends on the value of the Boolean or numeric scalar you wire to the external side of the selection terminal or *selector* . If a Boolean is wired to the selector, the structure must have two cases, False and True. If a numeric is wired to the selector, the structure can have from 0 to 2^15-1 cases. If you wire an enumeration to the selector, there must be one subdiagram for each enumeration item. The case identifiers are the same as the enumeration item names. If the structure does not have the appropriate number of cases, the VI in which it appears will be broken.

To add more cases, follow the procedure outlined in the Adding Subdiagrams topic. (If you wire a floating-point number to the selector, LabVIEW rounds it to the nearest integer value. LabVIEW coerces negative numbers to zero and values higher than the highest-numbered case to the last case.)

Note: Case statements in other programming languages generally do not execute any case if a case value is out of range. If you do not want out-of-range values to activate the highest or lowest cases in LabVIEW, you must either pretest the selector data for out-of-range numbers, or include a trap case that does nothing for out-of-range values.

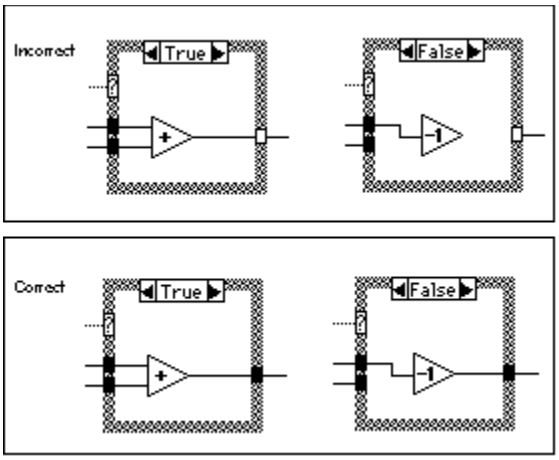
You can position the selector anywhere along the left border, but you must wire the selector. The selector

automatically adjusts to the data type. If you change the value wired to the selector from a numeric to a Boolean, cases 0 and 1 change to FALSE and TRUE. If other cases exist (2 through `n`), LabVIEW does not discard them, in case the change in data types is accidental. However, you must delete these extra cases before the structure can execute.

The same principle holds if you wire an enumeration to the selector and there are more cases than items in the enumeration. The diagram identifier for such cases is displayed as a grayed out numeric identifier to indicate that these cases must be deleted before the structure can execute.

The data at all input terminals (tunnels and selection terminal) is available to all cases. Cases are not required to use input data or to supply output data, but *if any case supplies output data, all must do so*. If you do not wire data to an output tunnel from every case, the tunnel turns white, as in the top example in the following illustration, and the run button shows a broken arrow.

When all cases supply data to the tunnel, it turns black, as in the bottom example in the following illustration, and the run button appears unbroken.



Adding, moving, and deleting Case subdiagrams are discussed after the following section on the Sequence Structure, because these operations are similar for both structures.

Sequence Structure

The Sequence Structure, which looks like a frame of film, consists of one or more subdiagrams, or *frames*, that execute sequentially. For an example of a VI that uses a Sequence Structure, see `examples\general\structs.llb\TimingTemplate.vi`.

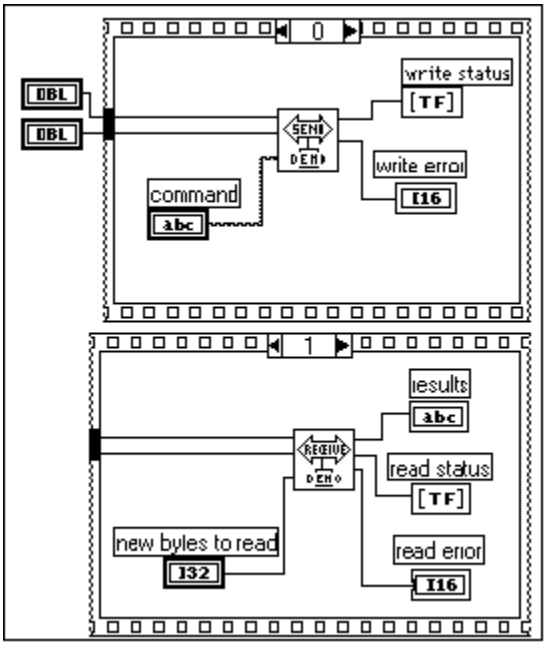
Determining the execution order of a program by arranging its elements in sequence is called *control flow*. BASIC, C, and most other programming languages have inherent control flow, because statements execute in the order in which they appear in the program. The Sequence Structure is the LabVIEW way of obtaining control flow within a dataflow framework. A Sequence Structure executes frame 0, followed by frame 1, then frame 2, until the last frame executes. Only when the last frame completes does data leave the structure.

Within each frame, as in the rest of the block diagram, data dependency determines the execution order of nodes.

You use the Sequence Structure to control the order of execution of nodes that are not data-dependent. A node that gets its data directly or indirectly from another node has a data dependency on the other node and always executes after the other node completes. You do not need to use the Sequence Structure when data dependency exists or when the execution order is unimportant.

If one part of a block diagram must execute before another part can begin, but data dependency does not exist between them, the Sequence Structure can establish the correct execution order. This situation

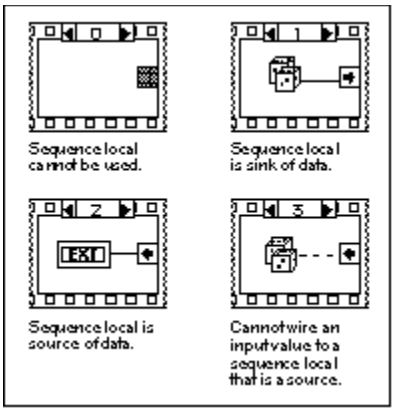
occurs often with GPIB applications. You may need to write a command to an instrument before taking a reading; however, the Receive VI does not use data from the Send VI and thus has no data dependency on it. Without the Sequence Structure, the Receive VI may execute first, causing errors. By placing the Send VI in frame 0 of a Sequence Structure and the Receive VI in frame 1, as shown in the following illustration, you can enforce the proper execution order.



Output tunnels of Sequence Structures can have only *one* data source, unlike Case Structures. The output can emit from any frame, but keep in mind that data leaves the structure only when it completes execution entirely, not when the individual frames finish. Data at input tunnels is available to all frames, as with Case Structures.

To pass data from one frame to any subsequent frame, use a terminal called a *sequence local*. To obtain a sequence local, choose **Add Sequence Local** from the structure border pop-up menu. This option is not available if you pop up on a sequence local or over the subdiagram display window. You can drag the terminal to any unoccupied location on the border. Use the **Remove** command from the sequence local pop-up menu to remove a terminal.

An outward-pointing arrow appears in the sequence local terminal of the frame containing the data source. The terminal in subsequent frames contains an inward-pointing arrow, indicating that the terminal is a source for that frame. In frames before the source frame, you cannot use the sequence local, and it appears as a dimmed rectangle. The following illustrations show the sequence local terminal.



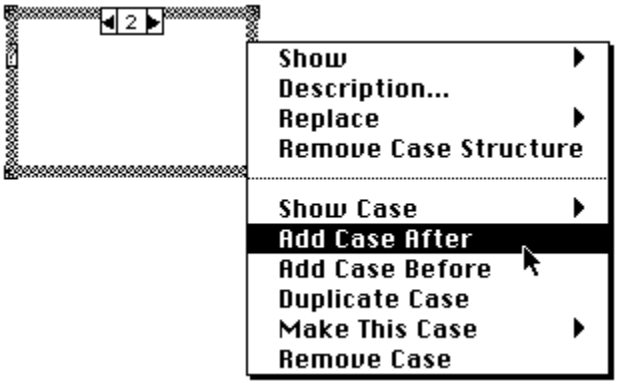
Editing Case and Sequence Structures

Because editing and manipulating the Case and Sequence Structures involves similar techniques, the following examples show only the Case Structure and its pop-up menus. For Sequence menus, substitute the word *frame* where the word *case* appears. A new Case Structure has two cases but can have one case as well. A new Sequence Structure has one frame.

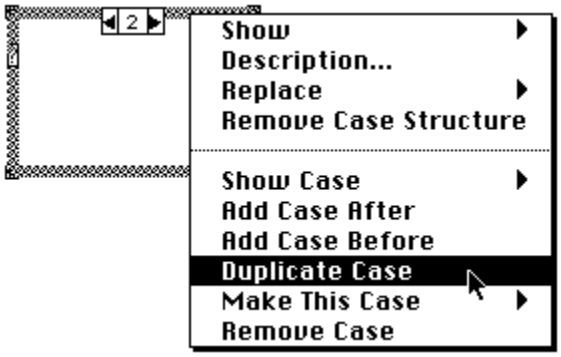
Adding Subdiagrams

You can add subdiagrams several ways.

If you select the **Add Case After** command from the structure border pop-up menu, as shown in the illustration that follows, LabVIEW inserts an empty subdiagram after the currently visible one. For example, an empty subdiagram at position 3 is created from subdiagram 2 using **Add Case After**. A complementary command, **Add Case Before**, appears in the same pop-up menu. These commands also appear in the diagram identifier pop-up menu.



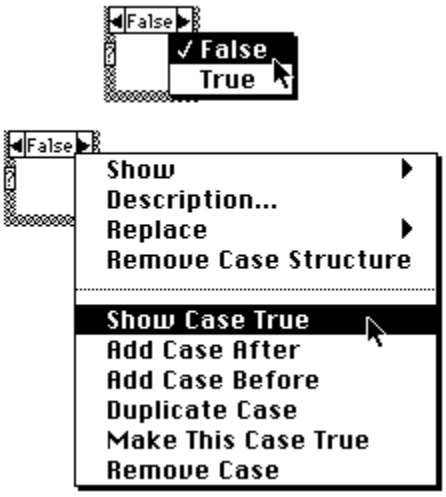
If you select **Duplicate Case** from either the structure border pop-up menu or the diagram identifier pop-up menu, as shown below, a copy of the visible subdiagram is inserted after itself.



When you add or remove subdiagrams, LabVIEW automatically adjusts the diagram identifiers to accommodate the inserted or deleted subdiagrams.

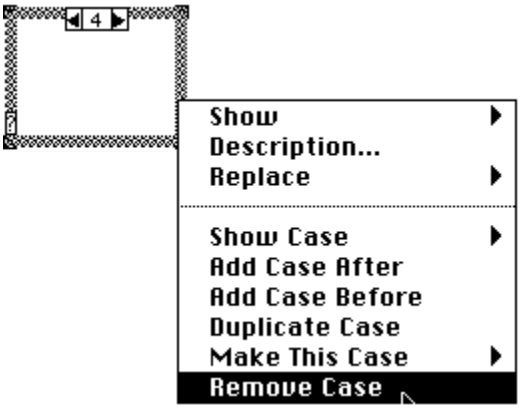
Moving between Subdiagrams

The fastest way to view the next lower or higher subdiagram is to click on the increment or decrement button in the display window. If you want to jump over several subdiagrams, click on the subdiagram identifier and select the destination subdiagram from the pop-up menu, as shown in the following illustration. You can also use the **Show Case** command from the border pop-up menu.



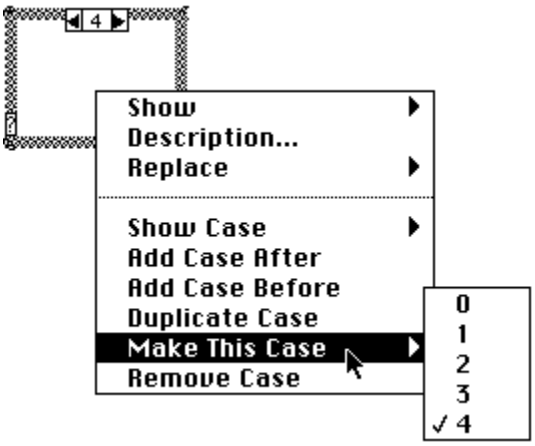
Deleting Subdiagrams

To delete the visible subdiagram, choose **Remove Case** from the structure border pop-up menu, shown below. The values of higher numbered subdiagrams adjust automatically. This command is not available if only one subdiagram exists.



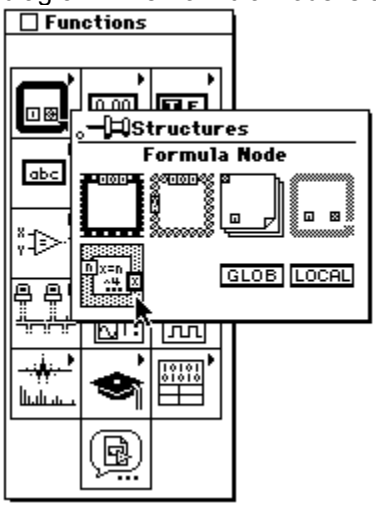
Reordering Subdiagrams

To move a subdiagram to another location, select **Make This Case** from the structure border pop-up menu, as shown below, and choose a new value from the hierarchical menu that appears. LabVIEW automatically adjusts the diagram identifiers of other subdiagrams.


4
Show
Description...
Replace
Show Case
Add Case After
Add Case Before
Duplicate Case
Make This Case
Remove Case
0
1
2
3
✓ 4

Formula Node

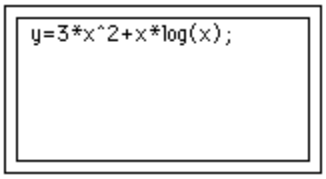
This topic describes how to use the Formula Node to execute mathematical formulas on the block diagram. The Formula Node is available from the **Structures** palette of the **Functions** palette.



Formula Node Use
Formula Node Functions
Formula Node Syntax
Errors Detected by the Formula Node

Formula Node Use

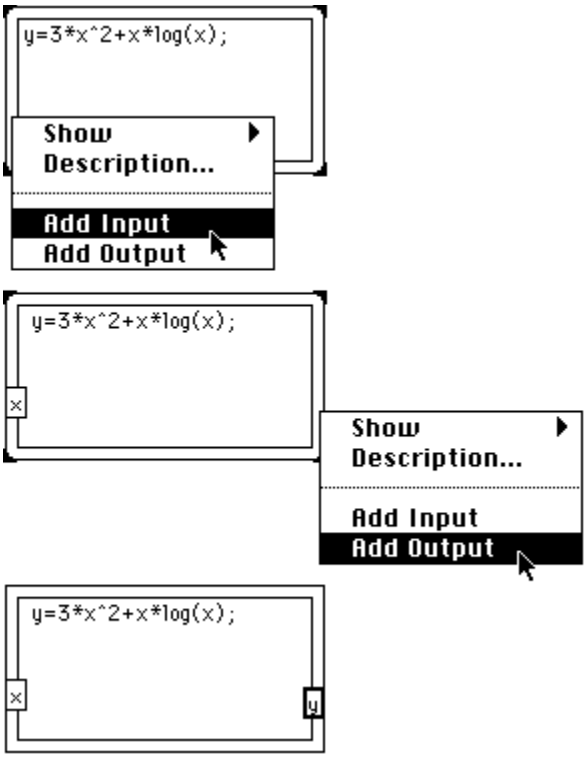
The Formula Node is a resizable box similar to the four structures (Sequence Structure, Case Structure, For Loop, and While Loop). Instead of containing a subdiagram, however, the Formula Node contains one or more formula statements delimited by a semicolon, as in the following example.



Formula statements use a syntax similar to most text-based programming languages for arithmetic expressions. You can add comments by enclosing them inside a slash-asterisk pair (/*comment*/).

See `examples\general\structs.llb\Equations.vi` for an example of a VI that uses a Formula Node.

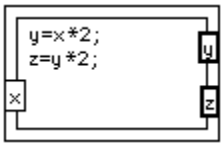
The pop-up menu on the border of the Formula Node contains options to add input and output variables, as shown in the illustration that follows.



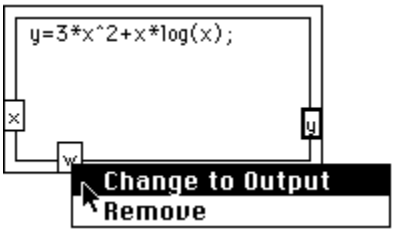
Output variables are distinguished by a thicker border.

There is no limit to the number of variables or formulas in a Formula Node. No two inputs and no two outputs can have the same name. However, an output can have the same name as an input.

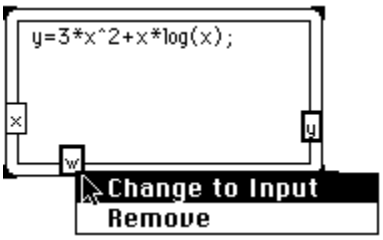
Every variable used in the Formula Node for a calculation must be declared as either an input or an output. Intermediate values, that is outcomes of operations calculated after input(s) to the node but before the final output from the node, must be declared as outputs. (However, intermediate values do not need to be wired to nodes external to the Formula Node.) For example, in the following illustration the y variable and the z variable must both be declared as outputs.



You can change an input to an output by selecting **Change to Output** from the pop-up menu, as shown in the following illustration.



You can change an output to an input by selecting **Change to Input** from the pop-up menu, as shown below.



All variables are floating-point numeric scalars, whose precision depends on the configuration of your computer. Variables cannot have units. All input variables that appear in the formulas must be wired. All output variables that are wired must be assigned in at least one statement; that is, they must be on the left side of an equal sign. An output variable may appear in an expression on the right side of an equal sign, but LabVIEW does not check to see whether it has been assigned in a previous statement. When an assignment occurs as a subexpression, the value of the subexpression is the value assigned; for example

```
x = sin (y = pi/3)
```

assigns (`pi/3)` to y, and then assigns [sin (`pi/3)]` to x.

If a syntax error occurs you can click on the broken run button to get the error listing. In the listing, the Formula Node displays a portion of the formula with a # symbol marking the point at which the error was detected.

Formula Node Syntax

The Formula Node syntax is summarized below using Backus-Naur Form (BNF) notation. Square brackets enclose optional items.

```
<assignlst>          = <outputvar> = <aexpr> ;
                          [ <assignlst> ]
<aexpr>              := <expr> | <outputvar> = <aexpr>
<expr>               := <expr> <binaryoperator> <expr>
                     |  <unaryoperator> <expr>
                     |  <expr> ? <expr> : <expr>
                     |  ( <expr>)
                     |  <inputvar>
                     |  <outputvar>
                     |  <const>
                     |  <function> ( <arglist> )
<binaryoperator>     := + | - | * | / | ^ | != | ==
                     | > | < | >= | <= | && | ||
<unaryoperator>      := + | - | !
<arglist>            := <aexpr>  [ , <arglist> ]
<const>              := pi | <number>
```

The precedence of operators is as follows, from lowest to highest.

=	assignment
? :	conditional

||	logical or
&&	logical and
!= ==	inequality, equality
< > <= >=	other relational: less than, greater than, less than or equal, greater than or equal
+ -	addition, subtraction
* /	multiplication, division
+ - !	unary: positive, negative, logical not
^	exponentiation

Exponentiation and the assignment operator are right-associative (groups right to left). All other binary operators are left-associative. The numeric value of TRUE is 1 and FALSE is 0 (for output). The logical value of 0 is FALSE, and any nonzero number is TRUE. The logical value of the conditional expression

```
<lexpr> ? <texpr>: <fexpr>
```

is `<texpr>` if the logical value of `<lexpr>` is TRUE and `<fexpr>` otherwise.

Errors Detected by the Formula Node

Error Message	Error Message Meaning
syntax error	Misused operator, and so on.
bad token	Unrecognized character.
output variable required	Cannot assign to an input variable.
missing output variable	Attempt to assign to a nonexistent output variable.
missing variable	References a nonexistent input or output variable.
too few arguments	Not enough arguments to a function.
too many arguments	Too many arguments to a function.
unterminated argument list	Formula ended before argument list close parenthesis seen.
missing left parenthesis	Function name not followed by argument list.
missing right parenthesis	Formula ended before all matching close parentheses seen.
missing colon	Improper use of conditional ternary operator.
missing semicolon	Formula statement not terminated by a semicolon.
missing equals sign	Formula statement is not a proper assignment.

Formula Node Functions

All function names must be lowercase.

Function	Corresponding LabVIEW Function Name	Description
abs(x)	Absolute Value	Returns the absolute value of x.
acos(x)	Inverse Cosine	Computes the inverse cosine of x in radians.
acosh(x)	Inverse Hyperbolic Cosine	Computes the inverse hyperbolic cosine of x in radians.
asin(x)	Inverse Sine	Computes the inverse sine of x in radians.
asinh(x)	Inverse Hyperbolic Sine	Computes the inverse hyperbolic sine of x in radians.
atan(x)	Inverse Tangent	Computes the inverse tangent of x in radians.
atanh(x)	Inverse Hyperbolic Tangent	Computes the inverse hyperbolic tangent of x in radians.
ceil(x)	Round to +Infinity	Rounds x to the next higher integer (smallest int ³ x).
cos(x)	Cosine	Computes the cosine of x in radians.
cosh(x)	Hyperbolic Cosine	Computes the hyperbolic cosine of x in radians.
cot(x)	Cotangent	Computes the cotangent of x in radians (1/tan(x)).
csc(x)	Cosecant	Computes the cosecant of x in radians (1/sin(x)).
exp(x)	Exponential	Computes the value of e raised to the x power.
expm1(x)	Exponential (Arg) - 1	Computes the value of e raised to the x power minus one (ex-1).

floor(x)	Round to -Infinity	Truncates x to the next lower integer (largest int # x).
getexp(x)	Mantissa & Exponent	Returns the exponent of x.
getman(x)	Mantissa & Exponent	Returns the mantissa of x.
int(x)	Round To Nearest integer	Rounds its argument to the nearest even integer.
intrz(x)	Round Toward 0	Rounds x to the nearest integer between x and zero.
ln(x)	Natural Logarithm	Computes the natural logarithm of x (to the base e).
lnp1(x)	Natural Logarithm (Arg +1)	Computes the natural logarithm of (x + 1).
log(x)	Logarithm Base 10	Computes the logarithm of x (to the base of 10).
log2(x)	Logarithm Base 2	Computes the logarithm of x (to the base 2).
max(x,y)	Max & Min	Compares x and y and returns the larger value.
min(x,y)	Max & Min	Compares x and y and returns the smaller value.
mod(x,y)	Quotient & Remainder	Computes the remainder of x/y, when the quotient is rounded toward -Infinity.
rand( )	Random Number (0- 1)	Produces a floating-point number between 0 and 1 exclusively.
rem(x,y)	Remainder	Same as mod except quotient is rounded to the nearest integer.
sec(x)	Secant	Computes the secant of x radians (1/cos(x)).
sign(x)	Sign	Returns 1 if x is greater than 0, returns 0 if x is equal to 0, and returns -1 if x is less than 0.

sin(x)	Sine	Computes the sine of x radians.
sinc(x)	Sinc	Computes the sine of x divided by x radians (sin(x)/x).
sinh(x)	Hyperbolic Sine	Computes the hyperbolic sine of x in radians.
sqrt(x)	Square Root	Computes the square root of x.
tan(x)	Tangent	Computes the tangent of x in radians.
tanh(x)	Hyperbolic Tangent	Computes the hyperbolic tangent of x in radians.
x^y	xy	Computes the value of x raised to the y power.

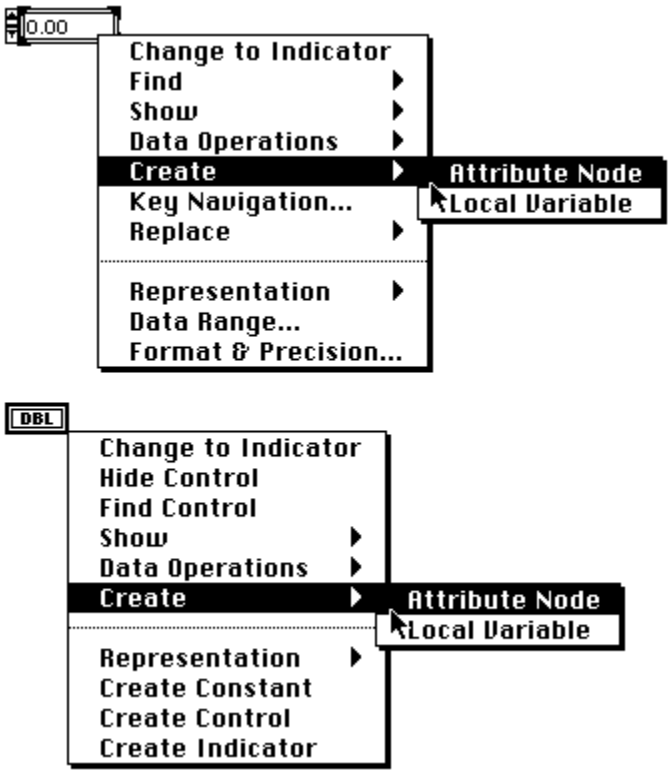
Attribute Nodes

This topic describes how to use Attribute Nodes to programmatically set and read attributes of front panel controls. Some useful attributes include display colors, control visibility, menu strings for a ring control, graph or chart scales, and graph cursors.

For more information about Attribute Nodes, see the following topics:

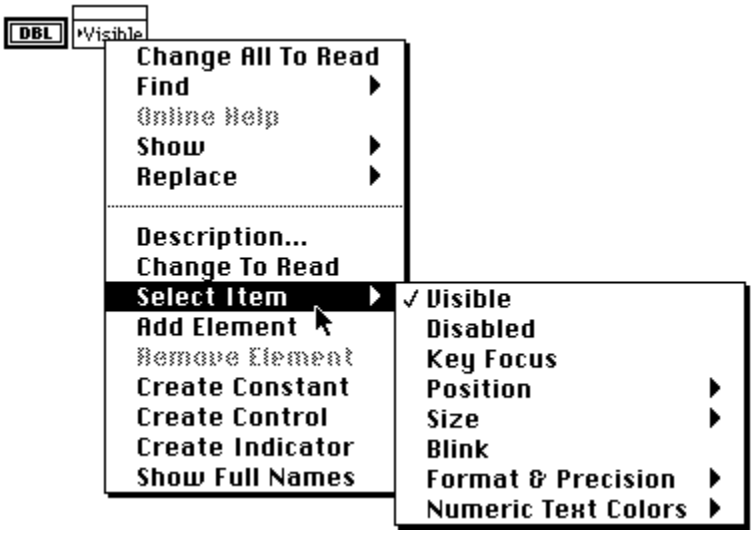
Attribute Node Usage
    Attribute Help
    Attribute Examples
Available Attributes

You create an Attribute Node by selecting the **Create»Attribute Node** option from the pop-up menu of a front panel control or from the controls terminal.



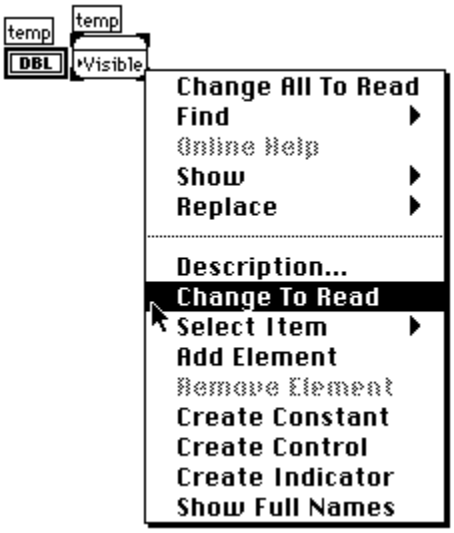
Selecting this option creates a new node on the diagram located near the terminal for the control, as shown in the following illustration. If the control has a label associated with it, the control label is used for the initial label of the Attribute Node. You can change the label after the node has been created.

If you pop up on an attribute terminal, and then choose Select Item, you get a menu of attributes which you can set for or read from the control, as shown in the following illustration. A shortcut to the list of attributes is to click on the Attribute Node with the Operating tool.

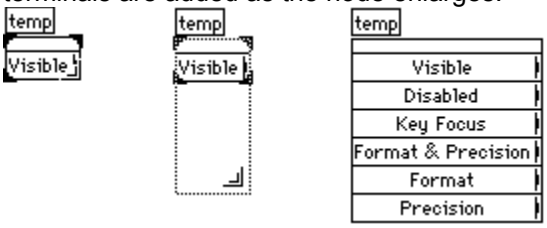


You choose whether to read or set attributes by selecting either the **Change to Read** or **Change to Write** option from the Attribute Node pop-up menu, as shown in the illustration that follows.

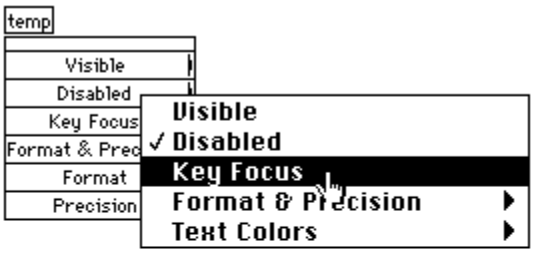
You can set an attribute when the small direction arrow is located on the left side of the terminal. You can read an attribute when the arrow is located on the right side of the terminal.

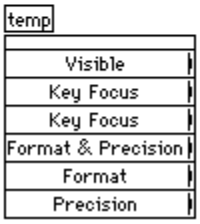


You can read or set more than one attribute with the same node by enlarging the Attribute Node. New terminals are added as the node enlarges.

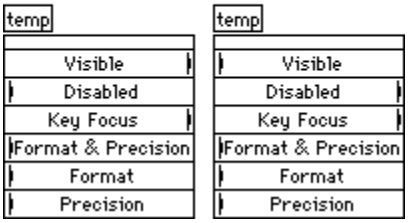


You associate a terminal with a given attribute by clicking with the Operating tool on the terminal and selecting an attribute from the Attribute Node pop-up menu.





You can create more than one Attribute Node by cloning an existing node, or by selecting the CreateÈAttribute Node option again. To clone an existing node, click and drag with the Positioning tool while holding down the <Ctrl> (Windows); <option> (Macintosh); <meta> (Sun); or <Alt> (HP-UX) key. Each node can have both read and write attribute terminals. However, you cannot copy and paste an Attribute Node using the **Edit** menu commands.



See examples\general\attribute.llb for an example of how to use an Attribute Node.

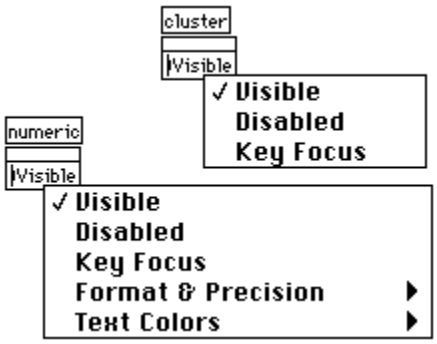
Attribute Node Usage

This topic consists of the following subtopics:

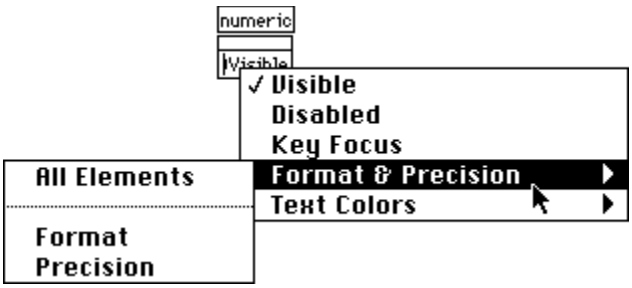
Attribute Help
Attribute Examples
- Setting the Strings of a Ring Control
- Selectively Presenting the User with Options
- Reading Cursors Programmatically

Available Attributes

You can create Attribute Nodes for any control on a front panel. If the control is an ARRAY or CLUSTER, only attributes relating to the overall ARRAY or CLUSTER are available. You can create Attribute Nodes for any of the controls in a CLUSTER and for the control in an ARRAY by selecting **Create»Attribute Node** from the subcontrol pop-up menu. The attributes for a CLUSTER and for a numeric control inside the CLUSTER are shown in the following illustration.



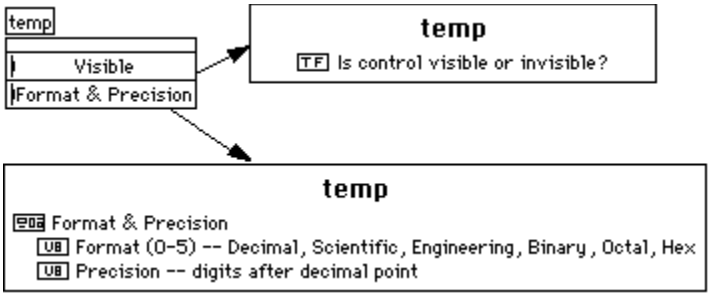
Some controls, such as the graph, have a large number of attributes you can read or set. Some of these attributes are grouped into categories and listed in submenus, such as the **Format & Precision** category for a numeric control. You can choose to set all of the attributes at once by selecting the **All Elements** option of the submenu. You can also set one or more of the elements individually by selecting the specific attribute(s). The **Format & Precision** option on a numeric control is shown in the following illustration as an example.



After you create an Attribute Node, the **Find Control** and **Find Terminal** options of the terminal and control pop-up menus change to submenus that help you find Attribute Nodes. In the same way, the Attribute Node has options to find the control and the terminal it is associated with.

Attribute Help

If you do not know the meaning of a given attribute, you can use the Help window to find out about the meaning of the attribute, its datatype, and acceptable values. If the attribute is a more complicated type, such as a CLUSTER, then the Help window displays a hierarchical description of the data structure. The following illustrations show an Attribute Node and the corresponding help information that is displayed as you move the cursor over different attribute names.



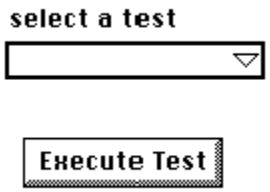
Attribute Examples

Included in your distribution software are example programs which illustrate how to programmatically set the scales of a graph, how to read and set the position of a cursor, and how to set the color table of an intensity graph. See the `examples` directory to explore these uses of the Attribute Node.

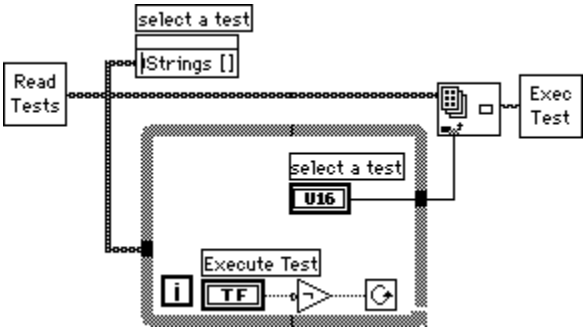
Setting the Strings of a Ring Control

The ring control is a pop-up menu control that holds the numeric value of the currently selected item. You can use it to present the user with a list of options. If the options cannot be determined until run-time, you can use the Attribute Node to set the options. See Ring Controls for more information about ring controls.

In the following example, the user is presented with a panel that has a ring control which displays a list of tests. The user selects a test and then presses the **Execute Test** button to continue.



The block diagram shown in the illustration that follows reads a list of valid tests from a file and passes the list, represented as an ARRAY of strings, to an Attribute Node for the ring control. The diagram then loops, waiting for the user to click on the **Execute Test** button. This gives the user a chance to select a test from the ring control, or to fill in information for other controls. When the user selects a test, the string corresponding to the numeric value of the ring control is read and then passed to a VI which executes the test.



Selectively Presenting the User with Options

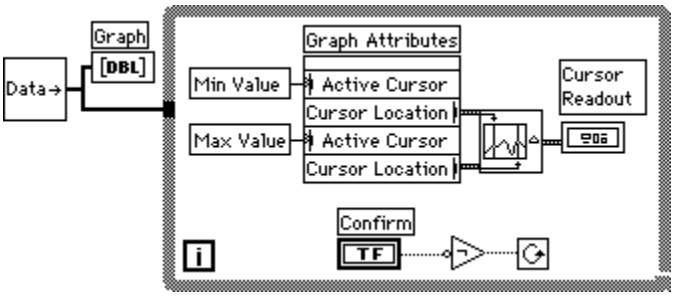
As users make selections, you may want to present them with other options. There are three possibilities.

1. One option is to use pop-up subVIs. You can create subVIs which have the options that you want to present to the user. By using the **Show Front Panel when Called** and **Close Afterwards if Originally Closed** options of **VI Setup...»Execution Options** when you create your subVI, you can have one of these subVIs open when called.
2. Another method for presenting options is to use the **Visible** option of Attribute Nodes to selectively show and hide controls.
3. The third method for presenting options is to use the **Disabled** option of Attribute Nodes to selectively disable controls. When a control is disabled, the user cannot change the value.

Reading Cursors Programmatically

You can use attributes to access information from one plot on a multi-plot graph, or one thumb on a multi-thumb slide, but you must specify which item is being operated on. The multiple cursors on a graph provide a good example of an attribute that must be activated before it can be accessed from the diagram.

The following block diagram shows a VI that displays data in a graph and programmatically reads the position of graph cursors.



Working through a While Loop, the VI first activates the Min Value cursor, and reads its numerical value. Next, the VI activates and reads the Max Value cursor. Then the VI calculates and displays information about the cursor selection on the front panel. When you press the **Confirm** button, the VI exits the loop and confirms the cursor positions.

See the examples in `examples\general\graphs\zoom.llb` for an application of programmatically reading graph cursors.

Available Attributes

When you want to select an attribute as a CLUSTER, for example Format & Precision, you must select **All Elements** in the attribute submenu.

ARRAY Attributes
Base Attributes
Boolean Attributes
CLUSTER Attributes
Color Box Attributes, Additional
Color Ramp Attributes
Digital Numeric Control Attributes
Fill Control Attributes, Additional
Intensity Chart Attributes
Intensity Graph Attributes
List Box Controls Attributes
Path Attributes
Refnum Attributes
Ring Attributes
Rotary, Slide, and Fill Control Attributes
String Attributes
Tables Attributes
Waveform and XY Graph Attributes
Waveform Chart Attributes

Available Attribute Descriptions

Base Attributes

A base set of attributes is available for all controls. These attributes include whether a control is visible, whether it is disabled, and its key focus. Remember that you can read these attributes as well as set them. For example, you can make a control invisible, and you can check to see if it is currently invisible.

TF **Visible.** Visible when True; hidden when False.
U8 **Disabled**

0: Control is enabled.
1: Control is disabled.
2: Control is grayed out.

Key Focus. When True, the control is the currently selected key focus. This means the cursor is active in this field. Key focus is generally changed by tabbing through fields.

Blinking. When true, blinks the control using the colors specified in the Preferences dialog box.

Position. Position of the top left corner of the control.

Top. Pixel position of the top of the control.

Left. Pixel position of the left side of the control.

Bounds. Dimensions of a bounding box containing the control and all of its parts, including the label, legend, scale, and so on. This is a read-only attribute because its main use is to read the overall size of a control so you can place it with respect to other controls. Each control has additional attributes that can be used to change the size of a control, such as Text Size, Button Size, Plot Area Size, and Number of Rows.

Width. Pixel width of the entire control.

Height. Pixel height of the entire control.

Attributes for Digital Numeric Controls

In addition to the base attributes, digital numeric controls have the following attributes:

Pixel Width. Width of the numeric text.

Format & Precision. CLUSTER of the format and precision attributes.

Format. The format for a numeric can be one of the following values.

0 - Decimal
1 - Scientific
2 - Engineering
3 - Binary
4 - Octal
5 - Hexadecimal
6 - Relative Time

Precision. Precision is the number of digits displayed after the decimal point.

Text Colors. Color numeric value describing the color for the text.

Text Color.

Background Color.

Attributes for Rotary, Slide, and Fill Controls

In addition to the base attributes, rotary, slide, and fill controls have the following attributes:

Housing Size. The size of the non-moving part that the slider moves on or over.

Width in pixels.

Height in pixels.

Active Slider. Which slider/needle is active.

Slider Colors. Active slider/needle color.

Foreground color for active slider/needle.

Background color for active slider/needle.

Scale Information

Scale Style. (0-8) sets scale style as illustrated in control pop-up menu from top left to bottom

right.

Format & Precision. CLUSTER of the format and precision attributes.

Format. The format for a numeric can be one of the following values.

0 - Decimal
1 - Scientific
2 - Engineering
3 - Binary
4 - Octal
5 - Hexadecimal
6 - Relative Time

Precision. Precision is the number of digits displayed after the decimal point.

Range. Minimum value, maximum value, and increment.

Minimum. Minimum value on slide/knob scale.

Maximum. Maximum value on slide/knob scale.

Increment. Distance between text markers.

Flipped. For vertical scales, Flipped is True if the minimum is at the top and the maximum is at the bottom. For horizontal and rotational scales (found on knobs, gauges, and so on), Flipped is True if the minimum is to the right and the maximum is to the left.

Mapping Mode. Linear or logarithmic.

Editable. Determines whether the scale can be edited by the user when the VI is running.

Additional Attributes for Fill Controls

In addition to the base attributes, and the attributes that fill controls have in common with rotary and slide controls, fill controls have the following attributes:

Fill Style. Sets fill style.

- 0 = no fill
- 1 = fill to min
- 2 = fill to max
- 3 = fill to value below
- 4 = fill to value above

Fill Color. Fill color for active slider.

Additional Attributes for Color Boxes

In addition to the base attributes, color boxes have the following attributes:

Color Area Size. The size of the colored area.

Width in pixels.

Height in pixels.

Attributes for Rings

In addition to the base attributes, rings have the same attributes as digital numeric controls. You can set or read the strings of the ring as an ARRAY of strings.

Ring Text Size Size of the text area of the ring.

Width in pixels.

Height in pixels.

Format & Precision. CLUSTER of the format and precision attributes.

Format. The format for a numeric can be one of the following values.

0 - Decimal
1 - Scientific
2 - Engineering
3 - Binary
4 - Octal
5 - Hexadecimal
6 - Relative Time

Precision. Precision is the number of digits displayed after the decimal point.

Text Colors. Color numeric value describing the color for the text in the digital display of the ring.

Text Color. Color numeric value describing the foreground color of the text.

Background Color . Color numeric value describing the color of the background on which the text is drawn.

Strings. ARRAY of the names of the items.

String

Attributes for List Box Controls

In addition to the base attributes, list boxes have attributes you can use to set the selection mode and the keyboard mode, and show or hide the scrollbar and the symbols column.

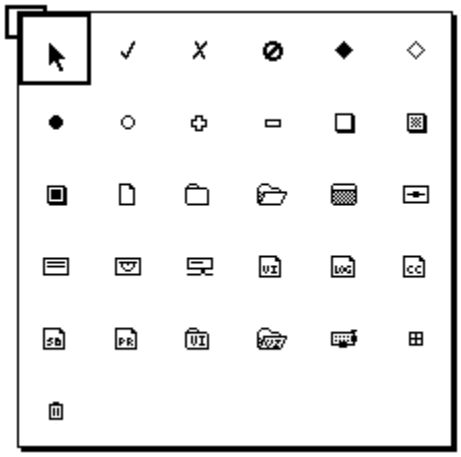
You use the Top Row attribute to read or set which row is showing at the top of the list box. You use the Num Rows attribute to read or set the number of rows visible at a given time and to set the width of each row.

The Double-Click attribute is a read-only attribute that indicates which item is the double-click item. This value is set whenever a user double- clicks on an item or types `<Enter>` (Windows and HP-UX), or `<Return>` (Macintosh and Sun) after selecting an item. The double-click value is -1 if nothing has been double-clicked. It is reset back to -1 after it is read using the Attribute Node. It is also reset to -1 if the user selects a different item, or if you set any of the other attributes.

You use the Item Names attribute to read or set an ARRAY of strings that are the options for the list box.

You use the Item Symbols attribute to read or set an ARRAY of 32-bit integers which indicate the symbols drawn next to each item. Notice that the option to view symbols must be turned on using either the **Show** pop-up menu option or the Symbols? attribute. The list box supports a limited number of built-in symbols; you cannot create an arbitrary symbol. The easiest way to select a symbol is to use the Listbox Symbol Ring constant, which is in the **Functions»Numeric»Additional Numeric Constants** palette.

After you place the Listbox Symbol Ring constant on a diagram, you can select a symbol from the symbol palette. Click on the Listbox Symbol Ring using the Operating tool to bring up a palette of options, shown in the following illustration, and select a symbol from this palette.



This constant is numeric. The constant associates a numeric value with each picture. Selecting the top-left corner item (blank) creates a constant of value zero. For example, selecting the checkmark creates a constant of value one.

Use the Disabled Items attribute to specify which options can be selected. It is an ARRAY of 32-bit integers, where each 32-bit integer is the number of a row you want to disable.

To set the Item Symbols attribute, create an ARRAY of these values.

Number of Rows. Height of list area in rows.
Pixel Width. Width of list area in pixels.
Selection Mode. The number of items that may be selected.

0 = zero or one
1 = one
2 = zero or more
3 = one or more

Keyboard Mode. Mode for selecting item by typing:

0 = System default
1 = Case-sensitive
2 = Case-insensitive

Scrollbar Visible. Is scrollbar showing (True) or hidden (False?).
Symbols Visible. Are symbols showing (True) or hidden (False?).
Top Row. The row number currently at the top of the list box.
Double-click. The last item that was double-clicked (read-only).
Item Names. The ARRAY of item names.
Item name.
Item Symbols. The ARRAY of item symbols.
Item symbol (0-22): use the Listbox Symbol Ring constant to assign values.
Disabled Items. ARRAY of indices of disabled items.
Width. Width in pixels of the list area.

Attributes for the Color Ramp

The color ramp has the same attributes as digital numeric controls. It has the same attributes for the scales as the slide controls. It also has attributes for whether the ramp and digital display are visible. You

can read or set the out of bounds colors (low and high), and you can select whether the control should interpolate between the specified values.

Color Area Size. The size of the colored area.
Width in pixels.
Height in pixels.
Numeric Text Colors. Color numeric value describing the color for the text in the digital display of the color ramp.
Foreground Color. Color numeric value describing the color of the foreground for the text.
Background Color. Color numeric value describing the color of the background for the text.
Ramp Visible. Is ramp visible?
Digital Display Visible. Is digital display visible?
Scale Info. Color scale information.
Interpolate Color. Interpolate between ARRAY colors?
Low Color. For values less than the first ARRAY.
High Color. For values greater than last ARRAY.
Color ARRAY. Color ARRAY elements.
Value/Color Pair
Value. Where the color is to be used.
Color
Style. Scale Style
Format & Precision. CLUSTER of the format and precision attributes.
Range. Minimum value, maximum value, and increment.
Flipped. Flipped is True if the minimum is at the top and the maximum is at the bottom.
Mapping Mode. Linear or logarithmic.
Editable. Determines whether the scale can be edited.

Attributes for Booleans

In addition to the base attributes, you can set the Boolean text strings that are displayed inside of the Boolean, and the colors of each of the states. The strings are an ARRAY of four elements, with the first element for the FALSE string and the second element for the TRUE string.

The other two strings only apply to Booleans that have a mechanical action, such as **Switch when Released** or **Latch when Released**. These behaviors are typically used in dialog boxes, where a button does not change status unless you release the mouse with the cursor inside the button. As you press the button, it highlights in temporary transition states, FALSE to TRUE and from TRUE to FALSE (the third and fourth states). As with the strings, the colors are an ARRAY of four elements.

If you pass an ARRAY of only two strings to the text strings Attribute Node, the first string is copied to the third string and the second string to the fourth string. If you pass an ARRAY of one string to the node, that string is copied to all four strings.

For example, assume you have a Boolean set to a mechanical action of **Switch when Released**, and you set pass a string ARRAY of run, stop, stop?, and run?. In the FALSE state, the Boolean has a string of run. If you press the button, the button highlights and the string changes to run?. If you release the mouse with the cursor in the button, the button changes to TRUE, and displays the word stop. If you release outside of the button, the button returns to FALSE and displays the word run. Pressing on the button while it is TRUE displays the string stop?.

Button Size. Dimensions of the button.

Width. Pixel width of the button.
Height. Pixel height of the button.
Strings [4]. Strings for False, True, True Tracking, and False Tracking states.
Boolean Text
Colors [4]. Foreground and background colors for False, True, True Tracking, and False Tracking states.
Colors
Foreground Color
Background Color

Attributes for Strings

In addition to the base attributes, you can set the scroll position for a string (the line number displayed at the top of the string control, with 0 representing the first line) and the selection range (measured in characters, with 0 representing the first character of the string). You can also select the Display Style, and enable/disable the scrollbar.

Text Size. Dimensions of the string.
Width. Pixel width of the string.
Height. Pixel height of the string.
Scroll Position. Scroll to line *N* from top of screen.
Selection. Text selection.
Selection Start. Beginning character position.
Selection End. Ending character position.
Display Style. Can be normal, backslash codes, password, or hex dump.
Scrollbar. Is vertical scrollbar enabled?
Text Colors. Color numeric value describing the color for the text.
Text Color.
Background Color.

Attributes for Tables

In addition to the base attributes, tables have attributes for the control of index indicator visibility, scrollbar visibility, row and column header visibility, selection color, edit position, index values, setting the data selection range, changing row and column headers, reading from and writing attributes to specific cells, setting cell foreground and background color, number of rows and columns visible, width and height of cells, and blinking.

Number of Columns. Number of columns visible in the table.
Number of Rows. Number of rows visible in the table.
Index Visible. Show or hide the index digital displays.
Vert Scrollbar. Show or hide the vertical scrollbar.
Horiz Scrollbar. Show or hide the horizontal scrollbar.
Row Headers Vis. Show or hide the row headers.
Col Headers Vis. Show or hide the column headers.
Selection Color. Set the color used to draw the current data selection.
Edit Position. Set the row/column of the current text entry location. In order for a cursor to display in the current text entry location, you must set the key focus for the table to TRUE before selecting the edit position.

 **Row**
 **Column**
 **Index Values.** Set the index value of the top left visible cell.
 **Row**
 **Column**
 **Selection Start.** Set start of data selection range.
 **Row.** Row number.
 **Column.** Column number.
 **Selection Size.** Number of elements in data selection.
 **Row.** Size 0 specifies an insertion point.
 **Column.** Size 0 specifies an insertion point.
 **Row Headers[].** ARRAY of strings.
 **Header string**.
 **Column Headers[].** ARRAY of strings.
 **Header string**.
 **Active Cell.** Cell from/to which to read/write cell specific attributes. You must set the active cell before specifying foreground and background colors and cell size.
 **Row.** Row number. When using the value -1, Row specifies the header and not the data. The value -2 for Row specifies the entire Column. The value -2 for both Row and Column specifies all the data.
 **Column.** Column number. When using the value -1, Column specifies the header and not the data. The value -2 for Column specifies the entire Row. The value -2 for both Row and Column specifies all the data.
 **Cell FG Color.** Foreground color for cells.
 **Cell BG Color.** Background color for cells.
 **Cell Size.** Dimensions of the active cell.
 **Width.** Width in pixels. [Sets the width of the entire column.]
 **Height.** Height in pixels. [Sets the height of the entire row.]

Attributes for Paths

In addition to the base attributes, you can set the selection range just as you can with a string control or indicator.

 **Text Size.** Dimensions of the path string.
 **Width.** Pixel width of the string.
 **Height.** Pixel height of the string.
 **Selection.** Text selection.
 **Selection Start.** Beginning character position.
 **Selection End.** Ending character position.
 **Text Colors.** Color numeric value describing the color for the path text.
 Text Color.
 **Background Color**.

Attributes for ARRAYs

In addition to the base attributes, ARRAYs have attributes for the visibility of the index display, the current values of the index, and the selection range for the ARRAY (used in copy and paste).

Number of Columns. Number of columns elements displayed horizontally.
Number of Rows. Number of elements displayed vertically.
Index Visible
Index Values
Index of top left element displayed in ARRAY.
Selection Start []
Index
Selection Size []. Number of elements in ARRAY selection.
Size. Size 0 specifies an insertion point.

Attributes for CLUSTERs

In addition to the base attributes, CLUSTERs have the following attribute:

Box Size. Dimensions of the CLUSTER box.
Width Width in pixels.
Height Height in pixels.

Attributes for the Waveform Chart

In addition to the base attributes, the waveform chart has attributes for reading and setting the visibility of the legend, palette, scrollbar and numeric displays. It also has attributes for the colors of the chart paper (background) and grid, the X and Y scale information, the update modes of strip, sweep, and scope, and the chart history data.

You can also set attributes for each of the plots. Attributes for plots are applied to the active plot. You select the active plot by setting the active plot attribute to the number of a plot, with 0 representing the first plot. After you select a plot, you can read or set attributes for that plot, including the color, interpolation, point style, and line style of the plot.

Plot Area Size. Dimensions of the plotting area in pixels (content rectangle). In case of stacked charts, refers to plotting area of active plot.
Width.
Height.
Legend Visible. Is the chart legend visible?
Palette Visible. Is the chart palette visible?
Scrollbar Visible. Is the chart scrollbar visible?
Digital Display(s) Visible. Is the chart digital display visible?
Transpose ARRAY. Transpose data before plotting?
Plot Area Colors. Colors for plotting surface.
FG Color. Foreground color for plotting surface.
BG Color. Background color for plotting surface.
Grid Colors. Grid colors.
X Color. Color for X grid.
Y Color. Color for Y grid.
X Scale Info
Scale Fit. Fit x scale to data (0-2) never, once, always.
Xo and Delta X. Offsets multiplier for scaling data in x direction.
Xo. X-axis value for first point.

Delta X. Interval between points.


Style. X scale style.


Format & Precision. CLUSTER of the format and precision attributes for the X scale.


Range. X scale range values.


Minimum. X scale minimum range value.


Maximum. X scale maximum range value.


Increment. Increment between X markers.


Flipped. For vertical scales, Flipped is True if the minimum is at the top and the maximum is at the bottom. For horizontal scales, Flipped is True if the minimum is to the right and the maximum is to the left.


Mapping Mode. Linear or logarithmic.


Editable. Determines whether the scale can be edited.


Y Scale Info (See X Scale Info.)


Active Plot. Which plot is active.


Plot Name. Name of active plot.


Plot Info. Attributes of active plot.


Plot Color. Color of active plot.


Plot Interpolation. Interpolation for active plot [0, 1, 2, 3] as shown in the pop-up menu from left to right.


Point Style. Point style for active plot.


Line Style. Line style for active plot (0-4) as shown in the pop-up menu from top to bottom.


Fill/Pt. Color. Color of points of fill of active plot.


Line Width. (0-5) pixels. 0 is hairline width for printing.


Bar Plot Style. (0-None, 1-3 vertical bars, 4-6 horizontal bars as shown in pop-up menu.) Horizontal bars are available only on XY charts.


Fill To. Specify the base to which you want to fill. Can be zero, negative infinity, positive infinity, or another plot.


Update Mode. Chart update mode (0-2) Strip, Scope, Sweep.


History Data. History data for chart. Can be used to clear the chart programmatically at the start of a VI.


Value.

Attributes for the Waveform and XY Graphs

Nearly identical to the waveform chart, the waveform graph adds Smooth Update, and various cursor attributes.


Smooth Update. Produce smooth (double buffered) updates?


Active Cursor. Which cursor is active.


Cursor Info. Attributes for active cursor.


Cursor Name. Name of active cursor.


Cursor Color. Color of active cursor.


Cursor Style. Grid style for active cursor (0-8) as shown in the cursor pop-up menu from top left to bottom right.


Cursor Point. Point style for active cursor (0-16) as Style shown in the cursor pop-up menu from top left to bottom right.


Cursor Name Visible . Shows or hides the cursor name.


Allow Drag. Enables or disables user to drag cursor.

Cursor Locked. How is the active cursor locked to a plot? Can be not locked, locked to a plot, or snaps to a point.

Cursor Plot. Plot with which the active cursor is associated (for locked cursors).

Cursor Index. Index of point to which the active cursor is locked (for locked cursors).

Cursor Position. Position of active cursor.

Cursor X. Horizontal position of active cursor (for unlocked cursors).

Cursor Y. Vertical position of active cursor (for unlocked cursors).

Cursor List. Cursor information for all cursors.

Selected Cursors. Which cursors are currently selected?

Attributes for the Intensity Chart

In addition to the base attributes, the intensity chart attributes are a superset of the standard chart and the color ramp.

Plot Area Size. Dimensions of the plotting area in pixels (content rectangle).

Width.

Height.

Ramp Visible. Is the ramp visible?

Z Scale Info. Color (Z) scale information.

Color. Interpolate between ARRAY colors?

Low Color. Color for values less than the last ARRAY element.

High Color. Color for values greater than the last ARRAY element.

Color ARRAY

Value/Color Pair

Value . Where the color is to be used.

Color

Scale Fit. Fit Z scale to data (0-2) never, once, always.

Zo and Delta Z. Scaling formula to apply to data (Z=(Delta Z) * Z + Zo).

Zo.

Delta Z

Style. Z scale style.

Format & Precision. CLUSTER of the format and precision attributes for the Z scale.

Range. Z scale range values.

Flipped. Flipped is True if the minimum is at the top and the maximum is at the bottom.

Mapping Mode. Linear or logarithmic.

Editable. Determines whether the scale can be edited.

Ignore ARRAY. Use colormap instead of ARRAY elements.

Color Table. Colormap 256 colors.

Color.

Update Mode. Update mode for chart (0-2) Strip, Scope, Sweep.

History Data. Data history for chart. Can be used to clear the chart programmatically at the start of a VI.

Value.

Attributes for the Intensity Graph

Intensity graph attributes are identical to the intensity chart, except that there is no choice whether to update as a sweep, scope, or strip chart.

Attributes for Refnums

Refnums have the base attributes only.

Global and Local Variables

This topic describes how to define and use global and local variables.

You can use global variables to easily access a given set of values throughout your LabVIEW application. Local variables serve a similar purpose within a single VI.

Global and local variables are advanced topics in LabVIEW. Be sure to study this topic carefully before using them.

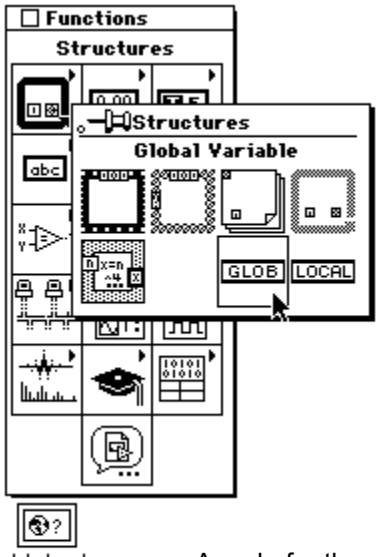
For examples of how to use global and local variables, see `examples\general\globals.llb` and `examples\general\locals.llb`.

Global Variables

A global variable is a built-in LabVIEW object. You define a global variZable by creating a special kind of VI, with front panel controls that define the data type of the global variable.

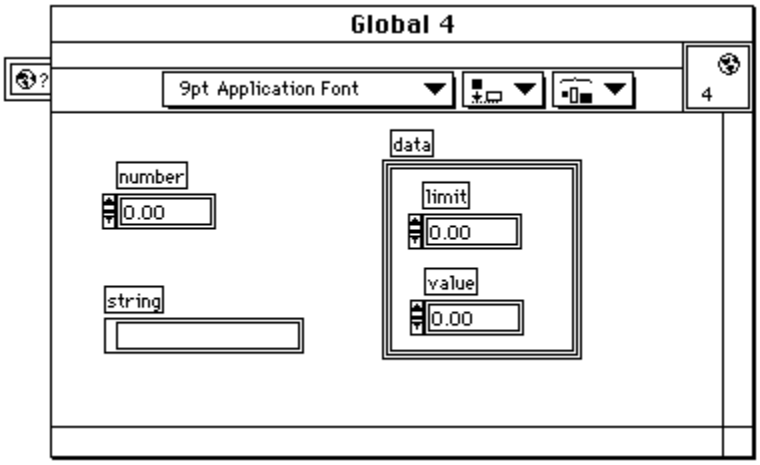
There are two ways to create multiple global variables. You can create several VIs, each with one item, or you can create one multiple global VI by defining multiple data types on the one global variable front panel. The multiple global VI approach is more efficient, and allows you to group related variables together.

You can create a global variable by selecting the global variable from the **Structures** palette of the **Functions** palette.

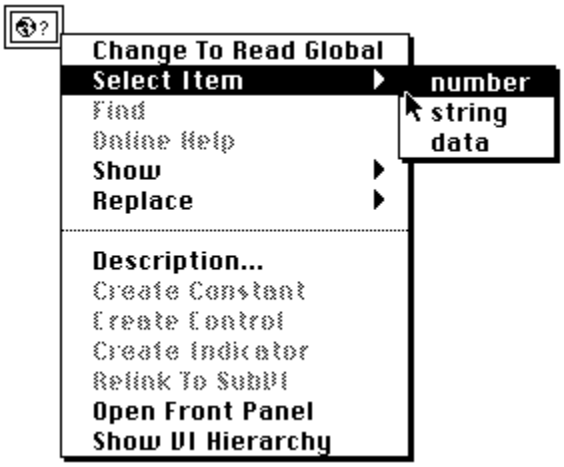


A node for the global appears on the block diagram. Double-click on the node to open a front panel. You use this panel to define the data type for one or more global variables. You should give each control a name, because you must refer to a specific global by name. After you have defined the data types for the global variables, save the VI.

The following illustration is an example panel describing three global types--a number, a string, and a cluster containing two values.



After you place a global on a diagram and define a panel for it, the node is associated with that panel. Because a global variable can define multiple globals, you must select which global you want to access from a given node. You select a global by popping up on the node and selecting the item by name from the **Select Item** menu as shown in the following illustration. Or you can click on the node with the Operating tool and select the item you want.



You can either write to a global variable or read from a global variable. Writing to a global variable means the value of the global is changed; reading from a global means the global is accessed as a data source. If you want to write to or read from a global, select the **Change to Write Global** or the **Change to Read Global** option of the global variable pop-up menu.

After you define a global variable, you can place it in other VIs using the **VI...** option of the **Functions** palette. If you select a global variable from the file dialog box, LabVIEW places the global variable node on the diagram. You can also clone, copy and paste, or drag-copy a global from the Hierarchy window.

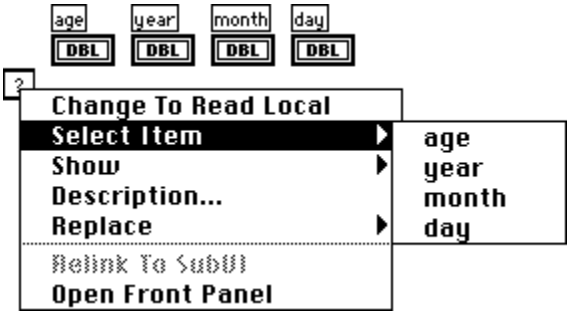
Local Variables

A local variable lets you read or write one of the controls or indicators on the front panel of your VI. Writing to a local variable has the same result as passing data to a terminal, except that you can write to it even though it is a control, or read from it even though it is an indicator. Also, you can have any number of local variables references to a given front panel control, with some in write mode, and others in read mode.

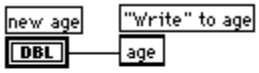
In effect, with a local variable reference you can use a front panel control as both an input and an output.

The easiest way to create a local is to pop up on the control or terminal and select **Create**»**Local**. A local is created on the block diagram. Another way to create a local is to select the local variable from the **Structures** palette of the **Functions** palette.

If there are no controls on the front panel of your VI, a node that looks similar to a global variable appears. If you have already placed controls on the front panel, the local variable node appears with the name of one of your controls showing.
You can pop up on the node, or click on it with the Operating tool to select the control you want to read or set from a list of the top-level front panel controls, as shown in the following illustration. You can also determine whether you want to read or write to the control by selecting either the **Change to Read Local** or the **Change to Write Local** options.



The following is an illustration showing how you can have multiple local variables accessing the same control, with each having a different sense, either read or write. You use the age variable twice in the diagram, once to write to, and another time to read from.



Be careful to sequence the access to local variables or global variables so that you get the results you want, as in the preceding example. There is no guarantee that the Write to age will occur before the Read from age if you do not create the proper sequencing yourself.

Custom Controls and Type Definitions

This topic introduces custom controls and type definitions.

You can customize a front panel control or indicator to make it better suited for your application. For example, you might want to make a Boolean switch that shows a closed valve when the switch is off and an open valve when it is on, a slide control with its scale on the right side instead of on the left, or a ring control with predefined text or picture items.

You can save a control or indicator that you have customized in a directory or VI library, just as you do with VIs. You can then use this control on other front panels. You can also create an icon for your custom control, and have the controls name and icon appear in the **Controls** palette.

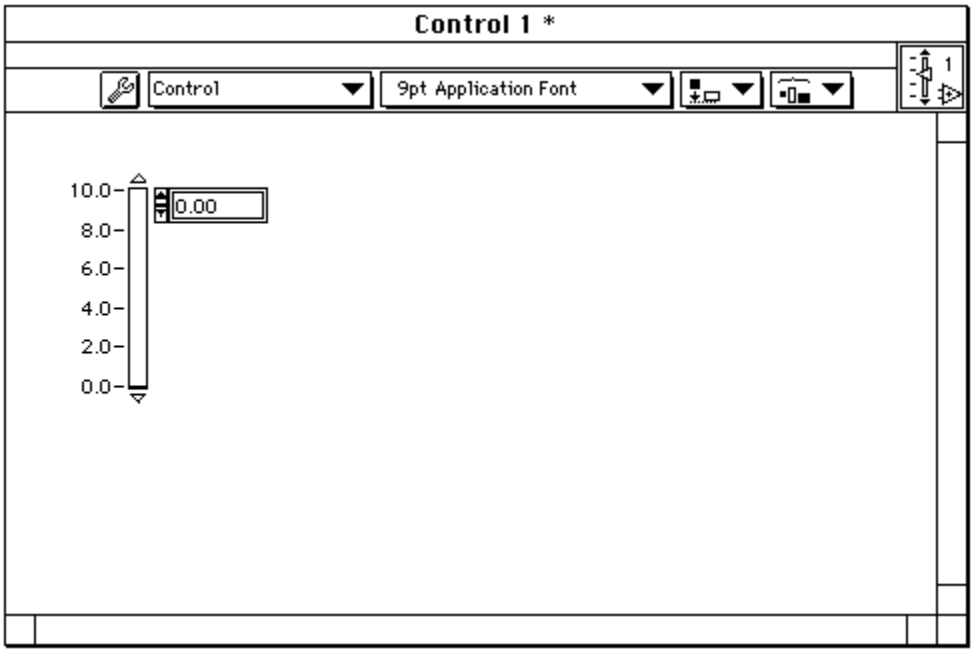
If you need the same control in many places in your VIs, you can create a master copy of that control, called a type definition. When you make a change to the type definition, LabVIEW automatically updates all the VIs that use it.

Custom Control Creation

Put your VI in edit mode to customize a control. On the front panel, place a control that is most like the one you want to create. For example, to create a slide with its scale on the right, start by placing any vertical slide on the front panel.

Select the Positioning tool. Select the slide control and choose **Edit**»**Edit Control...**. This option is available only when a control is selected. You can edit only one control from a panel at a time.

A window opens displaying a copy of the control. This window, shown in the following illustration, is called a Control Editor, and it is titled Control *N*, which is the name assigned to the Control Editor window until you save it and give it a permanent name.



A Control Editor window looks like a front panel, but it is used only for editing and saving a single control; it has no block diagram and cannot run.

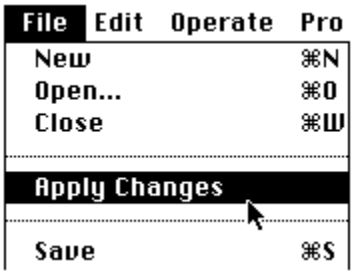
A Control Editor has an edit mode and a customize mode; the current mode is indicated by a button in the toolbar. A Control Editor is in edit mode when it first opens. In edit mode you can change the size or color of a control, and select options from its pop-up menu just as you do in edit mode of any front panel. In customize mode you can change the parts of a control individually.See the Customize mode topic for more information.
After you have edited a control, you can use it in place of the original control on the front panel that you were building when you opened the Control Editor. You can also save it to use on other front panels.

Applying Changes from a Custom Control
Custom Control Saving
Custom Control Usage
Making an Icon
Custom Controls are Independent from Source File

Applying Changes from a Custom Control

Custom Controls that are Valid

When you are ready to replace the original front panel control with your new custom control, select **Apply Changes** from the **File** menu of the Control Editor.



The File dialog box appears, asking you to name and select a location for your custom control. If your original front panel is the only place you use the custom control, you can close the Control Editor window without saving the control. Be sure to save the original VI with the custom control in place to preserve your work. If you want to use the custom control on other front panels in the future, you must save it as described in the Custom Control Saving topic.

Apply Changes is only available after you make changes to the control. **Apply Changes** is disabled if there is no original control to update. This happens when you delete or replace the original control, when you close the original front panel, or when you have opened a custom control that you saved earlier by selecting **File**»**Open**.

Custom Controls that are Valid

If the Control Editor has more than one control in it, the not-OK button appears, as shown above. A valid custom control must be a single control, though it may be a cluster of other controls. The not-OK button may appear temporarily while you move controls in and out of a cluster or array. To get an explanation for the error, click on the not-OK button. If there is more than one control in the Control Editor the error message reads There are extra objects on the front panel that do not belong to the custom control. If there are no controls the error message reads There must be one control on the front panel for a custom control to be valid. If you try to put a type definition control on the front panel of the Control

Editor, the error message reads You may not use a Type Definition in the control editor unless it is inside another control, such as a cluster or array. See the Type Definitions topic for more information.

Custom Control Saving

If you want to use your custom control on other front panels, choose **Save** from the **File** menu in the Control Editor window. You save a control the same way you save a VI, in a directory or in a VI library. A directory or VI library may contain controls, VIs, or both.

If you close the Control Editor window without saving your changes to the control, a dialog box asks you if you want to save the control.

Custom Control Usage

When you save your custom control, you can use it on other front panels by selecting **Control...** from the **Controls** palette on the front panel of any VI. Use the dialog box that appears to choose your control from the directory list and place it on your front panel.

For information about adding a custom control to the **Controls** palette, see Customizing the Controls and Functions Palettes.

Making an Icon

If you save your controls so that they appear in the **Controls** palette, you should make an icon representing the control before you save it. Pop up or double click on the icon square in the top right corner of the Control Editor window to create an icon for the control. This icon represents the control in the palette menu when you save the control in a library directory.

Custom Controls are Independent from Source File

You can open any custom control you have saved by selecting **File**»**Open**. A custom control always opens in a Control Editor window.

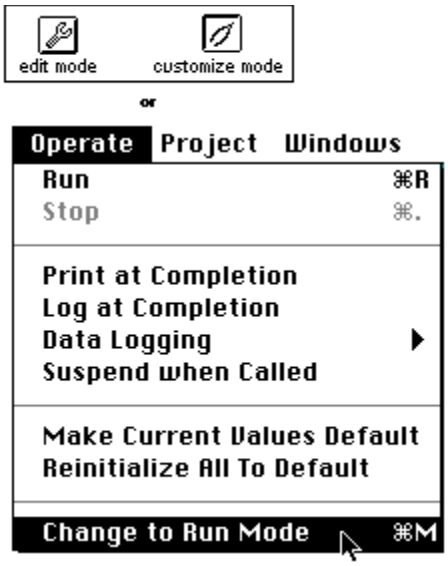
Changes you make to a custom control when you open it do not affect VIs that are already using that control. When you use a custom control on a front panel, there is no connection between that instance of the custom control and the file or VI library where it is saved; each instance is a separate, independent copy.

You can, however, create a connection between control instances on various VI front panels and the master copy of the control. To do this, you must save the custom control as a type definition or a strict type definition. Then, any changes you make to the master copy affect all instances of the control in all the VIs that use it. See the Type Definitions topic for more information.

Customize Mode

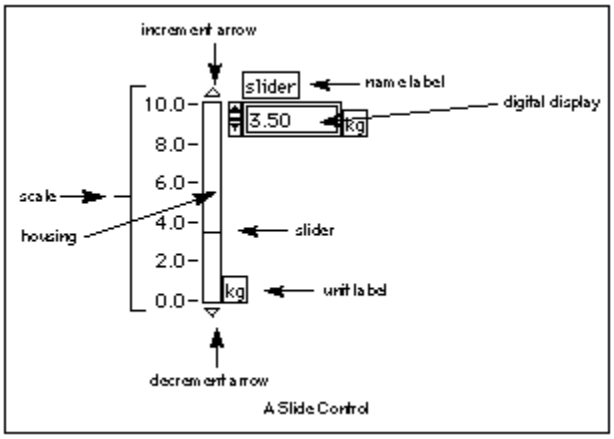
Independent Parts
Control Editor Parts Window
Customize Mode Uses Pop-Up Menus for Different Parts
Controls as Parts

You can make more extensive changes to a control in the customize mode of the Control Editor. Change between edit and customize mode by clicking on the mode button in the Control Editors toolbar, or by selecting **Change to Customize Mode** or **Change to Edit Mode** from the **Operate** menu while the Control Editor is the active window, as shown in the following illustrations.



Independent Parts

All LabVIEW controls are built from smaller parts. For example, a slide control consists of a scale, a housing, a slider, the increment and decrement arrows, a digital display, and a name label. The parts of a slide are pictured in the following illustration.



When you switch to customize mode in a Control Editor, the parts of your control become independent. You can make changes to each part without affecting any other part. For example, when you click and drag on the slides scale with the Positioning tool, only the scale moves. You can select parts and align or distribute them using the Alignment or Distribution rings from the toolbar; or change their layering order by selecting **Move Forward** or **Move Backward** from the **Edit** menu. Customize mode shows all parts of the control, including any that were hidden in edit mode, such as a name label or the radix on a digital control.

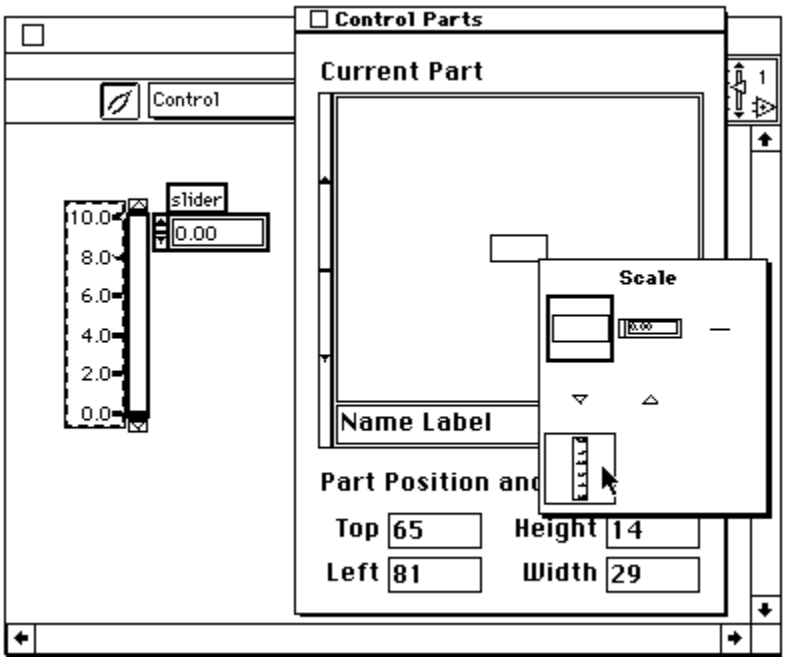
Because the parts of a control are detached from each other, you cannot operate or change the value of the control while in customize mode. Notice that the Operating tool is disabled. The Wiring tool is always disabled in a Control Editor because it has no use.

Control Editor Parts Window

When you are in the Control Editor, you can select **Windows»Show Parts Window**. The floating window that appears identifies the parts of the control, and shows you the exact position and size of each part. The *Current Part* display in the Parts window shows you a picture and the name of the part currently

selected in your Control Editor window. You can see a menu of all the parts by clicking on the current part display. You can also scroll through the parts of the control by clicking on the current part display increment or decrement arrow. When you change the part shown in the current part display, that part is selected on the control in the Control Editor window. When you select, change, or pop up on another part of the control in the Control Editor window, the part showing in the current part display also changes.

The illustration that follows shows the Control Editor window on the left overlaid by the Parts window on the right. The slides name label is the current part, and is selected in the Control Editor window. The Parts window shows the menu of parts that you get when you click on the current part display. This example shows that the name label is the current part, but that you are about to change to the scale part.



The Parts window shows you the exact position and size of the current part. These values are pixel values. When you move or resize a part in the Control Editor, the position and size in the Parts window are updated. You can also enter the position and size values directly in the Parts window to move or resize the part in the Control Editor. This is useful when you need to make two parts exactly the same size, or align one part with another. In the preceding illustration, the Parts window displays the position and size of the slides name label--the upper left corner of the label is at the pixel coordinates (45,63), and the label is 14 pixels high by 24 pixels wide.

The Parts window disappears if you switch to some other window. The Parts window reappears when you return to the Control Editor.

Customize Mode Uses Pop-Up Menus for Different Parts

Cosmetic Parts
Cosmetic Parts with More than One Picture
Cosmetic Parts with Independent Pictures

In customize mode, the pop-up menu for the control as a whole is replaced by a pop-up menu for each part. When you pop up on a part, you get a menu with some options available in edit mode, and some options available only in customize mode. Different parts have different pop-up menus.

There are three basic kinds of parts. Cosmetic parts, such as the slide housing, slider, and the increment and decrement arrows, are the most common. Cosmetic parts display a picture.

The second kind of part is a text part, such as the name label of the slide. Text parts consist of a picture for the background (usually just a rectangle) and some text.

Finally, a part may be another control. The slide, for example, uses a numeric control for a digital display. Knobs, meters, and charts also use a numeric control for a digital display. Some controls are even more complicated than that; for example, the graph uses an array of clusters for its cursor display part.

Cosmetic Parts

A cosmetic part is a picture. The following illustration shows a pop-up menu for a cosmetic part, such as a slide housing. To pop up on a cosmetic part, you must be in customize mode. You must pop up on the part itself, not on the picture of the part in the Parts window.

Copy to Clipboard puts a copy of the parts picture on the Clipboard. If you select **Copy to Clipboard** for the slide housing, the Clipboard contains a picture of a tall, narrow inset rectangle. This Clipboard picture can be pasted onto any front panel or imported as the picture for another part using **Import Picture**. These pictures are just like the **Decorations** in the **Controls...** menu.

When you need simple shapes like the housing rectangle for other parts, there are several advantages to using pictures copied from original LabVIEW parts, instead of making them in a paint program. Pictures taken from LabVIEW parts or decorations look better than pictures made in a paint program when you change their size. For example, a rectangle drawn in a paint program can only grow uniformly, enlarging its area but also making its border thicker. A rectangle copied from a part like the slide housing keeps the same thin border when resized.

Another advantage is that LabVIEW parts appear basically the same on both color and black-and-white monitors.

In addition, you can color pictures taken from LabVIEW parts or decorations with the LabVIEW Color tool. Pictures imported from another source keep the colors they had when they were imported, because those colors are a part of the definition of that picture.

Import Picture and **Import at Same Size** replace a cosmetic parts current picture with whatever picture is on the Clipboard. Use these options to individually customize the appearance of your controls. For example, you could import pictures of an open and closed valve for a Boolean switch.

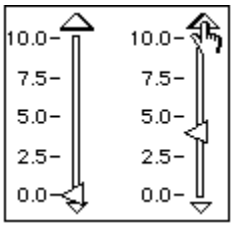
Import at Same Size replaces the current picture, but keeps the parts original size, shrinking or enlarging the Clipboard picture to fit. If there is no picture on the Clipboard, both **Import Picture** and **Import at Same Size** are disabled.

Revert restores the part to its original appearance. **Revert** does not change the parts position. If you opened the Control Editor window by selecting **Edit Control** from a front panel, LabVIEW reverts the part to the way it looks on that front panel. If you opened the Control Editor window by selecting File»Open, **Revert** is disabled.

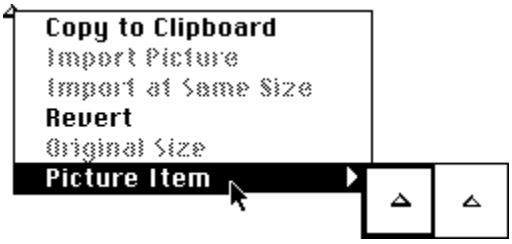
Original Size sets a parts picture to its original size. This is useful for pictures that you imported from other applications and then resized. Some of these pictures do not look as good as the original when resized, and you might want to restore their original size to fix them. If you have not imported a picture, **Original Size** is disabled.

Cosmetic Parts with More than One Picture

Some cosmetic parts have more than one picture, which they display at different times. These different pictures are all the same size and use the same colors. For example, the slides increment arrow is a picture of a triangle, normally raised slightly from the background. It also has another picture, a recessed triangle, that appears while you are clicking on it with the Operating tool to increment the value of the slide. The illustration that follows shows the two pictures of an increment arrow in action.



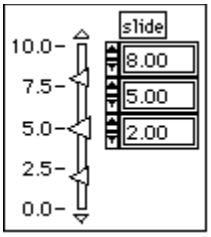
A cosmetic part with more than one picture has the **Picture Item** option on its pop-up menu, as shown in the following illustration.



Picture Item displays all the pictures belonging to a cosmetic part. The picture item that is currently being displayed has a dark border around it. When you import a picture, you change only the current picture item. To import a picture for one of the other picture items, first select that picture item and then import the new picture.

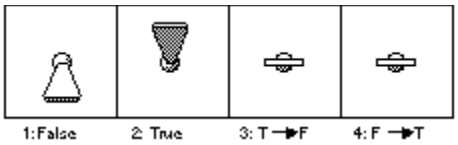
Cosmetic Parts with Independent Pictures

A cosmetic part with more than one picture can have pictures of different sizes, which each use different colors. The slide, for example, uses two pictures of different sizes to show which slider is active on a multi-value slide. The slide in the following example uses a bigger triangle to show that the middle slider is the active one.



A Boolean switch also has more than one picture. Each picture may be a different size and have different colors. A Boolean switch has four different pictures--the first shows the false state; the second shows the true state. You use the third and fourth pictures when you set the mechanical action of a Boolean control to either Switch When Released or Latch When Released.

Until you release the mouse button, the value of the Boolean does not change with these two mechanical actions. Between the time you click on the button and the time you release the click (while you are thinking, Do I really want to do this?), the Boolean shows the third or fourth picture as an intermediate state. The third picture is for the true to false transition state, and the fourth is for the false to true state. In the following illustration of a toggle switch, the third and fourth pictures are the same, but this is not always the case.



There is a shortcut in the Control Editor for importing pictures into a Boolean control. While in edit mode (you do not have to be in customize mode), you can select **Import Picture**»**True** or **Import Picture**»**False** from the pop-up menu of a Boolean. Doing this imports the picture into both the normal state and the corresponding transition state.

You can also import different pictures for the transition states in customize mode. To do this, first pop up on the button and use Picture Item to change the third picture. With the True -> False picture on the Clipboard, pop up again and select **Import Picture**. Repeat these steps for the fourth (False -> True) picture.

When a cosmetic part can have different sized pictures, the part has the **Independent Sizes** option on its pop-up menu, as shown in the following illustration.

Independent Sizes is an option you can turn on only while in customize mode if you want to move and resize each picture individually without changing the cosmetic parts other pictures. Normally, this option is not checked; and, when you move or resize the cosmetic parts current picture, its other pictures also move the same amount or change size proportionally.

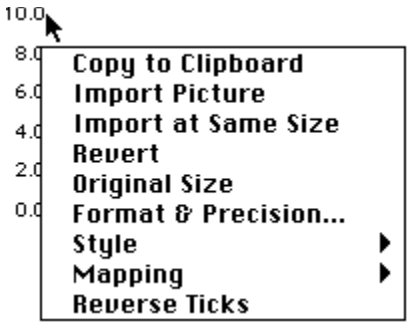
Text Parts

A text part is a picture with some text. The pop-up menu for a text part, such as a name label, has some items identical to those on a cosmetic parts pop-up menu. The other items on this menu are the same as the text pop-up menu in front panel edit mode. An example pop-up menu for a text part is shown in the following illustration.

Only the background picture for the text part is shown in the Parts window, not the text itself. The background picture can be customized, not the text.

Scale Parts

A scale is a special kind of text part with markers for the text. The scales picture is the background for each of its markers. This background is usually a transparent rectangle and therefore not visible, but you can see it if you color one of the scale markers. A scale has the same options on its pop-up menu as a text part, along with other options relating only to scales. This menu is shown in the following illustration.



Reverse Ticks changes the tick marks on a vertical scale from the right side to the left, or vice versa. On a horizontal scale, it changes the tick marks from the top to the bottom, or vice versa. On a rotary scale, such as the scale on a knob, dial, gauge or meter, it changes the tick marks from the outside to the inside, or vice versa. **Reverse Ticks** does not move the scale.

You can position the scale by dragging it while in customize mode. Hold the <Shift> key down when dragging a slides scale to restrict the movement to one direction only.

Controls as Parts

A control can include other controls as parts. A common example of this is the digital display on a slide, knob, meter, or chart. There is no difference between the digital display and the ordinary, front panel digital control, except that the digital display is serving as part of another control.

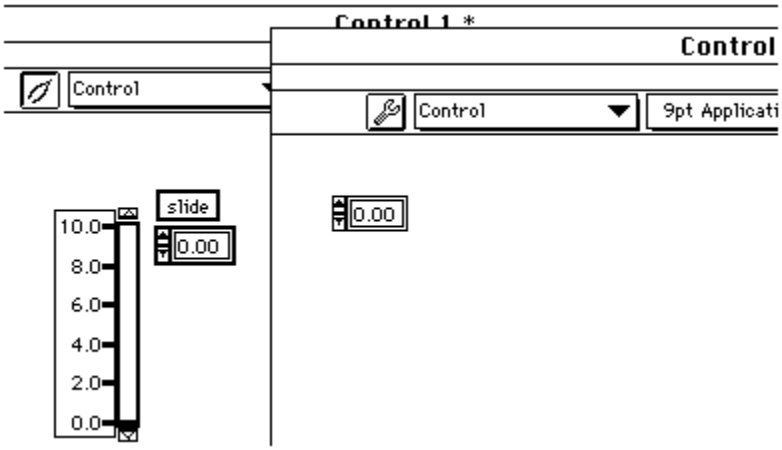
The digital display is also made up of parts. When you are editing the original control in the Control Editor, the digital display behaves as a single part, so you cannot change or move its parts individually. You can, however, open a Control Editor for the digital display and customize it there.

To customize a control that is a part of another control, open a Control Editor for it. You can open a Control Editor window for the part directly from the original front panel, if it can be selected separately from the main control in edit mode. The digital display can be selected separately from the slide control, for example. Then you can choose **Edit»Edit Control...**.

You can always open a Control Editor window for the part from the Control Editor window of the main control. Select the part in the Control Editor and choose **Edit Control**. Control editors can be nested in this way indefinitely, but most controls use other controls as parts only at the top level. An exception is the graph, which uses complicated controls as parts, which, in turn, use other controls as parts.

You cannot open a second Control Editor window for the main control already being customized.

The following illustration shows the Control Editor for the slide on the left, and a Control Editor window for the digital display on the right. You do not have to be in customize mode to open a nested Control Editor window unless you are unable to select the control part in edit mode.



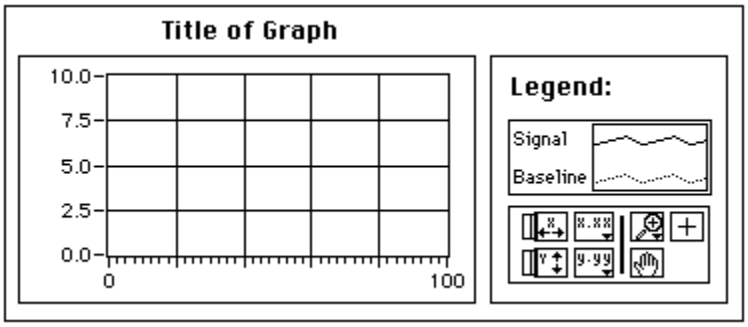
Cosmetic Parts Added to a Custom Control

When you are making a custom control in the Control Editor, you can make even more changes to its appearance by adding cosmetic or text parts to it.

If you paste a picture or text from the Clipboard, create a label with the Labeling tool, or select a picture from the **Decorations** palette, that picture or text becomes a part of your control and appears with the control when you place it on a front panel. You can do this in either edit or customize mode in the Control Editor. You can move, resize, or change the layering order of the new part, just like any other part. Your addition appears as a decoration part in the Parts window in customize mode.

You can also delete decoration parts while you are in the Control Editor.

The following illustration shows a custom graph that has some decoration parts, including the Title of Graph label, the Legend: label, and the box around the legend parts.



When you use a custom control on other front panels, you can change the size of any decoration parts that you added, but you cannot move them.

Custom Controls Caveats

There are some things you need to be aware of when you make custom controls. One is the difference between pictures on different platforms. Pictures created on one platform may look slightly different when loaded on another platform (this applies to pictures imported into a Pict Ring or used as background on any front panel as well). For example, a picture with an irregular shape or a transparent background might have a solid white background on another platform. See the Picture Differences topic for more information.

The Control Editor can change only the appearance of a control; it cannot change the behavior of a control. This has two implications. First, you cannot change the way a control displays its data. Second,

you cannot change the way a control behaves when you edit it, especially when you resize it.

For example, when you make a ring control taller, the increment and decrement arrows also increase in height. If you move the increment and decrement arrows so they are side by side at the bottom of the ring, the ring continues to make them become taller when it increases, and you get some strange results.

Be sure to experiment with your custom control after you make it, in case it has any odd behavior. If you like the control the way you have made it, but are not pleased with its irregular editing behavior, read about strict type definitions in the next section, Type Definitions, to learn how editing can be restricted.

Type Definitions

A *type definition* is a master copy of a control. You use the Control Editor to create the master copy, or type definition. Type definitions are useful when the same kind of control is used in many VIs. You can save the control as a type definition, and use that type definition in all your VIs. Then, if you need to change that control, you can update the single type definition file instead of updating the control in every VI that uses it.

Type Definition: Data types must match
Strict Type Definition: Everything must match
Type Definition Creation
Type Definition Usage
Type Definition Automatic Updating
Type Definition Searches
Cluster Type Definitions

Type Definition: Data types must match

A type definition forces the control data type to be the same everywhere it is used. Use a type definition when you want to use a control of the same data type in many places and when you may want to change that data type automatically everywhere it is used. For example, suppose you make a type definition from a double-precision digital control, and that you subsequently use that type definition in many different VIs. Later you change the type definition to a 16-bit integer digital control. LabVIEW automatically updates every VI that uses that type definition, or if you prefer, indicates to you that the type definition in a VI needs to be updated.

You can also make a type definition that is a cluster, such as a cluster of two integers and a string. If you change that type definition to a cluster of two integers and two strings, LabVIEW updates the type definition everywhere it is used.

As long as the data type matches the master copy, a type definition can have a different name, descriptions, default value, size, color, or even a different style of control (for example, a knob instead of a slide).

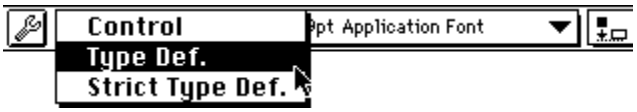
Strict Type Definition: Everything must match

A type definition can also force almost *everything* about the control to be identical everywhere it is used, not just its data type but also its size, color, and appearance. This is called a *strict type definition*.

The only aspects of a control that can be different from the master copy of a strict type definition are the name, the description, and the default value. As an example, suppose you make a strict type definition that is a double-precision digital control with a red frame. Like the type definition, if you change the strict type definition to an integer, LabVIEW automatically updates every VI that uses it, or indicate that they need to be updated. Unlike the type definition, however, other changes to the strict type definition, such as changing the red frame color to blue, also requires VIs using it to be updated.

Type Definition Creation

You make a type definition by setting the ring on the toolbar in a Control Editor window, as shown in the following illustration. Set up the control the way you want it, and choose **File»Save** in the Control Editor window.



You can open any type definition you have saved by selecting **File**»**Open**. A type definition always opens in a Control Editor window. Any changes you make to a type definition affect all VIs that are using it.

Type Definition Usage

Place type definitions and strict type definitions on the front panel of a VI as you would any custom control. You can edit and operate a type definition on your front panel as you would any other control.

Note: You cannot edit a strict type definition on your front panel except to change its name, description, or default value.

You can tell that a control is a type definition when you see the type definition options in its pop-up menu, as shown in the following illustration. You can recognize a strict type definition on your front panel because you cannot edit it, and most of its pop-up menu options are missing.

For each type definition that you use on a front panel, LabVIEW keeps a connection to the file or VI library in which it is saved. You can see this connection in action if you place a type definition on a front panel and then select it and choose **Edit Control** from the **Edit** menu. The Control Editor that opens is the type definition you saved, with the name you gave it, instead of the generic Control N.

Type Definition Automatic Updating

LabVIEW ensures that the data type is the same everywhere a type definition is used, and that everything about a strict type definition is the same everywhere it is used. LabVIEW automatically updates any type definitions or strict type definitions on your front panel that are incorrect, replacing the one on your front panel with an exact copy of the one saved in the file or VI library.

If you have edited an instance of a type definition on your front panel extensively, such as coloring and resizing it, you might not want this automatic update feature. You can pop-up on the type definition on your front panel and turn off the **Auto-Update from Type Def.** option. Instead of automatically updating this type definition when necessary, the VI has a broken run arrow and the type definition on the front panel is disabled. You cannot run the VI until you fix the type definition, either by selecting the option **Update from Type Def.** from the pop-up menu, or by changing the data type to match the type definition.

Whenever you use a type definition, you can give it a different default value. However, if the data type of the type definition changes, all default data is updated from the master copy. If the data type does not change, the individual default values can be preserved.

Type Definition Searches

Because LabVIEW must keep a connection to the type definition, the file or VI library containing the type definition must be available in order to run a VI using it. If you open a VI and LabVIEW cannot find a type definition that the VI needs, the type definition control on the front panel is disabled and the run arrow is

broken. To fix this problem, you must either find and open the correct type definition, or pop-up on the disabled control and select **Disconnect From Type Def.** Disconnecting from the type definition removes the restrictions on the controls data type and appearance, making it into an ordinary control. You cannot re-establish that connection unless you find the type definition and replace the control with it.

Cluster Type Definitions

If you use a type definition or strict type definition that is a cluster, it is a good idea to use the Bundle by Name and Unbundle by Name functions on the block diagram to access the clusters elements, instead of the Bundle and Unbundle functions. These functions reference elements of the cluster by name instead of by cluster order, and are not affected when you reorder the elements or add new elements to the cluster type definition. If you delete an element that you are referencing in Bundle by Name or UnBundle by Name, you have to change your block diagram. Refer to the description of these functions in Cluster Controls and Indicators.

Calling Code from Other Languages

This topic explains various methods of calling code written in other languages into LabVIEW. These methods include using platform-specific protocols; creating a Code Interface Node to call code written specifically to link to LabVIEW VIs' using a Call Library Function node to call Dynamic Link Libraries (DLLs) under Windows, Code Fragments on the Macintosh, and Shared Libraries on UNIX; and using the LabWindows/CVI Function Panel converter to convert an instrument driver written in LabWindows/CVI.

Executing Other Applications from within Your VIs
Call Library Function
LabWindows/CVI Function Panel Converter (not available on the Macintosh)

Executing Other Applications from within Your VIs

You can execute other applications from within your VIs. The methods are different on Windows and UNIX than on the Macintosh.

(Windows, UNIX) You use the System Exec VI to execute other applications from within your VIs. You can use the simple System Exec VI from the **Functions**»**Communication** palette to execute a command line from your VI. The command line can include any parameters supported by the application you plan to launch.

If you can access the application through TCP/IP (or DDE in Windows), you may be able to pass data or commands to the application. See the reference material for the application you plan to use to see the extent of its communication capability. You can also refer to Communications Overview, for more information on techniques for using networking VIs to transfer information to other applications.

(Macintosh) You use AppleEvents VIs to execute other applications from within your VIs. AppleEvents are a Macintosh-specific protocol through which applications can communicate with each other. They can be used to send commands between applications. You can also use them to launch other applications.

Call Library Function Use
Code Interface Node Use

Call Library Function Use

You can call most standard shared libraries (in Windows these are Dynamic Link Libraries or DLLs, on the Macintosh they are Code Fragments, and on UNIX they are Shared Libraries) using the Call Library Function node. The Call Library Function node includes a large number of data types and calling conventions. You can use it to call functions from most standard and custom-made libraries.

The Call Library Function node is most appropriate when you have existing code that you want to call, or if you are familiar with the process of creating a DLL in Windows, Code Fragments on the Macintosh, or Shared Libraries on UNIX. Because a library uses a format that is standard among several development environments, you should be able to use almost any development environment to create a library that LabVIEW can call. Check the documentation for the compiler you are using to see if it can be used to create standard shared libraries.

See the Call Library Function topic for a detailed description of this node.

Code Interface Node Use

For applications in which you need the highest performance, or you want to pass arbitrary data structures to C code, you can create a Code Interface Node (CIN). By using CINs, you can call code written specifically to link to LabVIEW VIs.

The CIN is a very general method for calling C code from LabVIEW. You can pass arbitrarily complex data structures to and from a CIN. In some cases, you may get higher performance using CINs because data structures are passed to the CIN in the same format they are in when stored in LabVIEW.

To get this level of performance, however, you must learn how to create a CIN. This requires that you be a good C developer and take sufficient time to create the CIN you need. Also, because CINs are more tightly coupled with LabVIEW, there are restrictions on which compilers you can use.

See the LabVIEW Code Interface Reference Manual for information on how to create Code Interface Nodes. The functions are available in help form. See your release notes about seting up Adobe Acrobat Reader for use with this file.

Call Library Function

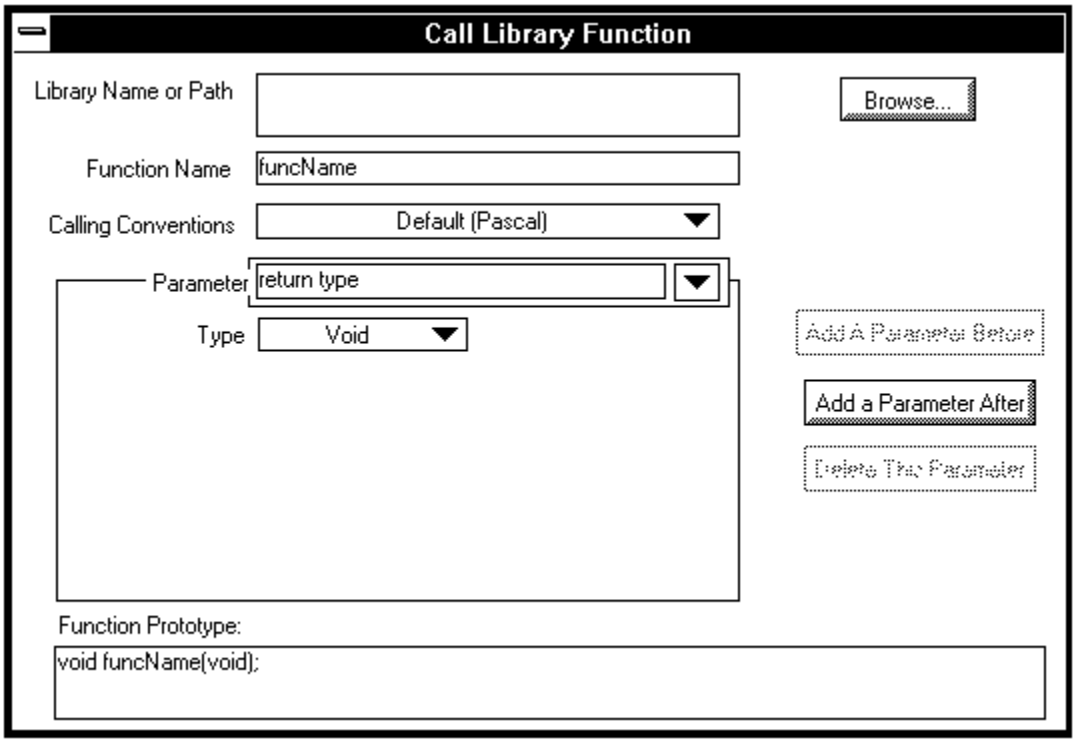
(Windows, UNIX) Using the Call Library Function node directly, you can call a Windows 3.1 16-bit DLL, a Windows 95 or Windows NT 32-bit DLL, a Macintosh Code Fragment, or a UNIX Shared Library function directly.

(Macintosh) The Call Library node uses the Macintosh's Code Fragment Manager (CFM). This is standard on all PowerMac machines. 680x0 Macintosh computers need the CFM extension. Additionally, the Call Library Node on 680x0 Macintosh computers cannot call variable argument functions. Shared libraries on the Macintosh operate differently than on other platforms. A file may contain more than one code fragment, each of which will have a name.

The Call Library Function node, shown in the following illustration, is available from the **Advanced** palette of the **Functions** palette.

If you double-click on the Call Library Function node or select **Configure...** from its node pop-up menu, LabVIEW displays a dialog box you can use to specify the library, function, parameters, and return value for the node (plus calling conventions in Windows). When you click the **OK** button, the node automatically adds the correct number of terminals, and sets the terminals to the correct data types.
The return value for the function returns to the right terminal of the top pair of terminals of the node. If there is no return value, this pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the functions parameter list. You pass a value to the function by wiring to the left terminal of a terminal pair. You read the value of a parameter after the function call by wiring from the right terminal of a terminal pair.

The Call Library Function dialog box is shown in the following illustration.



As you select options in the dialog box, an indicator at the bottom, called Function Prototype, displays the C prototype for the selected function.

Note: In Windows 3.1, an upper limit of 29 4-byte arguments can be passed via the Call Library Function. Double-precision floating point parameters passed by value are 8-byte quantities and therefore count as two arguments. You are thus limited to 14 double-precision floating point parameters passed by value.

Under Windows 3.1, DLLs must be 16-bit. Under Windows 95 and Windows NT, DLLs must be 32-bit. If you have a 16-bit DLL that you want to call from Windows 95 or Windows NT, you must either recompile it as a 32-bit DLL or create a "thunking" DLL. Refer to Microsoft documentation for information on thunking DLLs.

Calling Conventions (Windows)
Parameter List Creation
Calling Functions That Expect Other Data Types

Calling Conventions (Windows)

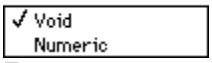
Use the calling conventions of the ring to select the calling conventions for the function. The default calling convention for Windows 3.1 is Pascal, and Standard C for Windows 95 and Windows NT. This default corresponds to the calling convention that is used by most DLLs. The alternative option is to use C calling conventions. Refer to the documentation for the DLL you are trying to call to determine which calling conventions you should use. Under Windows 3.1, almost all DLLs use Pascal calling conventions. Under Windows 95 and Windows NT, almost all DLLs use standard C calling conventions.

Parameter List Creation

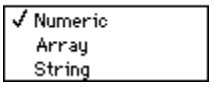
Initially, the Call Library Function node has no parameters, and has a return value of void. You can click on the **Add a Parameter Before** and **Add a Parameter After** buttons to add parameters to the function. You can click on the **Delete this Parameter** button to remove a parameter.

You can use the parameter ring to select different parameters or the return value. Once selected, you can change the parameter name to something more descriptive. The parameter name does not affect the call, but it is propagated to output wires. Descriptive names make it easier to switch between parameters.

Specify the type of each parameter using the type ring. The return type is limited to either Void, meaning the function does not return a value, or Numeric, shown in the following illustration.



For parameters, you can select Numeric, Array data types, or, String data types as shown in the following illustration. Additionally, you can receive a Void return value.



When you select an option from the type ring, you get more options you can use to specify details about the data type and how to pass the data to the library function. LabVIEW has a number of different options for data types, because of the variety of data types required by different libraries. Refer to the documentation for the library you call to determine which data types to use.

Numeric Data Types

For numeric data types, you must specify the exact numeric type using the data type ring. Options include the following.

- Signed and unsigned versions of 8-bit, 16-bit, and 32-bit integers
- Four-byte single-precision numbers
- Eight-byte double-precision numbers

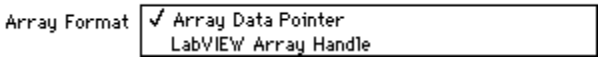
You cannot use extended-precision numbers and complex numbers because they are not generally used in standard libraries.

You also need to use the format ring to specify whether you want to pass the value or a pointer to the value.

Under Windows 3.1, you cannot use either single or double precision data types as the return type. This is because there is no standard method for DLLs to return single precision and double precision numbers, so this sort of return is implemented differently by each compiler that does it. If you need to return a single- or a double-precision number, pass the data back as a parameter instead of as the return value.

Array Data Types

You can specify the data type of arrays (using the same options as for numeric data types), the number of dimensions, and the format to use in passing the array. Use the **Format** option to select whether you want to pass a pointer to the array data, or pass a pointer to a LabVIEW array, which includes a four-byte value for each dimension followed by the data. If you select **Array Data Pointer**, shown in the following illustration, you will probably need to pass the array dimension as separate parameter(s).



Note: In Windows 3.1, LabVIEW uses only the array data pointer format for arrays. In addition, under Windows 3.1 you can specify that the data should be passed using a Huge pointer. You can use a Huge pointer if you need to pass more than 64 K. You should only turn this option on if the DLL you are calling expects a Huge pointer. If you try to pass a Huge pointer to a function that expects a normal pointer, LabVIEW may crash.

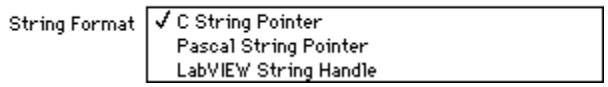
If the library function that you are calling is written specifically for LabVIEW, you should use the LabVIEW

array format, because it explicitly includes the size information. For most conventional libraries, however, you should use the array data pointer, and you will probably be expected to pass the size of the array as a separate parameter(s).

Caution: Do not attempt to resize an array with system functions such as realloc. Doing so may cause your system to crash.

String Data Types

You can specify the string format for strings. The options for string format are C string, Pascal string, or LabVIEW string. These are shown in the following illustration.



Base your selection of the string format on the type of string that the library function expects. Most standard libraries expect either a C string (string followed by a NULL character) or a Pascal string (string preceded by a length byte). If the library function that you are calling is written specifically for LabVIEW, then you may want to use the LabVIEW string format, which consists of four bytes for length information, followed by string data.

Caution: Do not attempt to resize a string with system functions such as realloc. Doing so may cause your system to crash.

In Windows 3.1, you cannot use the LabVIEW String Handle format.

Calling Functions That Expect Other Data Types

In some cases, you may encounter a function that expects a data type LabVIEW does not use. For example, you cannot use the Call Library Function node to pass an arbitrary cluster, or an array of non-numeric data.

Depending on the data type, you may be able to pass the data by creating a string or array of bytes that contains a binary image of the data that you want to send. You can create binary data by typecasting data elements to strings and concatenating them.

Another option is to write a library function that accepts data types LabVIEW does use, and parameters to build the data structures that the library function expects, then calls the library function.

Finally, you can write a code interface node instead. Code interface nodes can accept arbitrary data structures, but take some time to master because you have to understand how LabVIEW passes data.

LabWindows/CVI Function Panel Converter

Note: This feature is not available on the Macintosh.

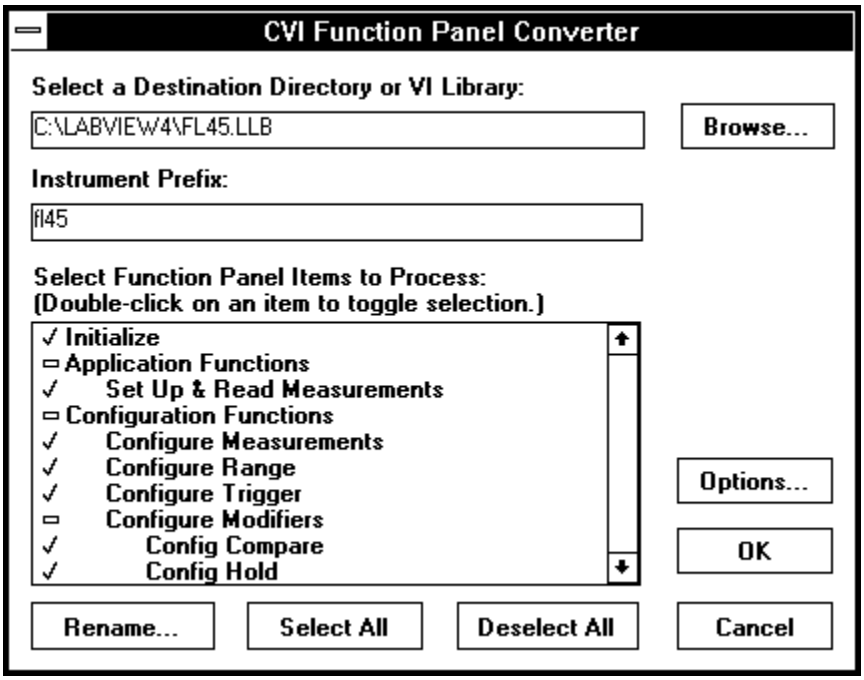
The LabWindows/CVI Function Panel converter automates the process of converting instrument drivers written in LabWindows/CVI so they can be used in LabVIEW. A LabWindows/CVI instrument driver consists of a C source and header files, a Dynamic Link Library (DLL) (Windows) or shared library (UNIX) containing the compiled code, and a CVI-specific file called Function Panel (FP) file. FP files are used in CVI to allow users to specify arguments and return values when editing a C function call statement by filling in values in a pop-up window.

The LabWindows/CVI Function Panel converter converts each function into a LabVIEW VI, using the information in the CVI FP file to determine the type, data representation, and placement of controls on the VI front panels. Each generated VI uses a Call Library Function node on its block diagram to call the appropriate C routine in the provided driver library.

While LabVIEW has many instrument drivers that are written completely on the diagram without calling library functions, the LabWindows/CVI Function Panel converter has advantages in some cases. For instruments where a LabVIEW driver isn't available, this tool makes it relatively easy to take advantage of a CVI driver if one exists. Given the option, however, a true LabVIEW driver consisting of LabVIEW diagrams is preferable because it is easy for customers to view and modify, and because it multitasks well within the LabVIEW environment. Library calls are synchronous, so any VIs running in parallel pause for the duration of the call.

LabWindows/CVI Function Panel Conversion Process

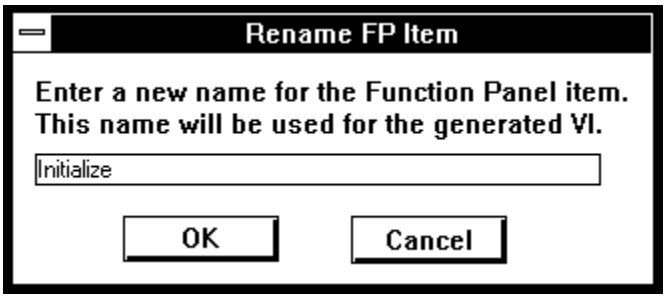
If you select **File**»**Convert CVI FP File...**, a dialog box appears asking you to select a LabWindows/CVI Function Panel file. When you have selected an FP file, you see the following dialog box, in which you can indicate where to save new VIs and what driver functions to convert.



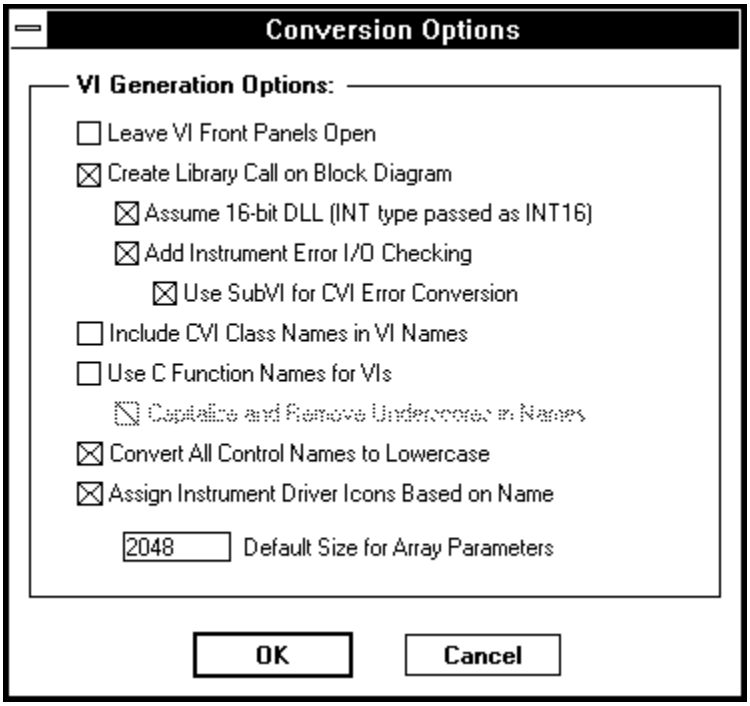
You specify the destination directory or VI library in the topmost text box. The suggested destination is shown, and you can change the path to place the new VIs that LabVIEW will create anywhere you want. A **Browse** button allows you to specify the destination via a file dialog box in the usual way.
The Instrument Prefix text box shows the instrument prefix as provided by the function panel file. This

prefix is prepended by CVI to all C function names in the FP tree. The converter likewise prepends this prefix to the names of VIs it generates. The suggested prefix is shown, based on the DLL, but you can change or delete this string as desired.
The remainder of the dialog box is devoted to the selection of function panels items to convert. A list box lists all items in the function panel tree, indented hierarchically by class, as in CVI. The class names are listed as well, but are grayed out and not selectable. Initially, all nodes found in the FP file are selected. Selected items are indicated by a checkmark symbol, non-selected items by no symbol, and class names by a rectangle.

Double-clicking an item toggles selection. The **Select All** and **Deselect All** buttons can be used for convenience. You can also single-click on an item to move the highlighted bar to that item and click **Rename**. This pops up the following simple dialog box for renaming an FP item:



The Options... button brings up the following dialog box:



The conversion options are as follows**:**

Leave VI Front Panels Open--Causes the converter to leave VIs in memory with their front panels open when conversion is complete rather than disposing of each VI after it is saved to disk. (Off by default.)

Create Library Call on Block Diagram--Causes the converter to place a Call Library Function node on the block diagram of each VI and wire up all front panel terminals appropriately. If this option is not on, only the front panel is created; nothing is dropped on the block diagram. (On by default.)

Assume 16-bit DLL--***(Windows)*** Causes the converter to treat the Integer type from the CVI function

panel as an INT16 rather than an INT32 when creating the type descriptor describing the arguments to the Call Library Function node. This is necessary when calling DLLs created with some third-party compilers such as Borland C. (This option is selectable only under Windows 3.1, where it defaults to on.)

Add Instrument Error I/O Checking--Causes the converter to drop error-handling code on the block diagram of each VI created. If you select this option, Error In and Out clusters are dropped on the front panel, positioned below all other controls and indicators derived from the CVI function panel. The front panel terminals and Call Library Function node on the block diagram are enclosed in a case structure which is executed only if the status field of Error In is FALSE, indicating no error. (On by default.)

Use SubVI for CVI Error Conversion--Drops a subVI on the block diagram of each VI in order to map CVI-style error codes to LabVIEW-style error clusters suitable for feeding to the General Error Handler. The subVI is copied from a template located in `vi.lib\instr\visatl.llb` into the destination VI library. The integer return value from the Call Library Function node, the Error In cluster, and the current VI's name is passed to the subVI. If an error is detected by the subVI, it is passed to Error Out, with status set to TRUE. If a warning is detected, it is passed to Error Out, but with status set to FALSE. Otherwise, Error In is passed to Error Out. If this option is not set, no subVI is dropped, and the block diagram is constructed so that an error is indicated in Error Out only if the Call Library Function return value is less than zero. (On by default.)

Include CVI Class Names in VI Names--Automatically creates a name for each VI based on either the function panel name or the C function name for each instrument driver option. If this option is checked, the Class name associated with the CVI function panel is prepended to the automatically generated name for the function.

Use C Function Names for VIs--Normally, the converter constructs the names of the VIs it generates directly from the function panel item names by merely prepending the instrument prefix and appending "`.vi`." Unfortunately, this approach does not always produce unique names for each item, because CVI does not require that the "leaves" of the function panel tree have unique names. Setting this option causes the converter to construct VI names from the actual C function names which the FP items correspond to, thereby guaranteeing unique results.

To avoid problems caused by duplicate names, the converter checks to see if all items are unique whenever it builds the list of function panel items to be displayed. Non-unique items are flagged with a ϕ symbol, and are not selectable by double-clicking. In order to prevent problems caused by duplicate names, the converter brings up a one-button alert whenever it first opens its dialog box if it detects such name conflicts, and automatically turns on the Use C Function Names option. If you prefer to use the function panel names instead, you can manually turn off this option, and then individually rename the items which have name conflicts. Renamed items retain their user- supplied names even when **the Use C Function Names** option is changed.

Capitalize and Remove Underscores in Names--Because names built from C function names tend to be less "pretty" than those derived from the function panel item names, this option attempts to make them "prettier" by capitalizing initial letters and replacing underscores with spaces. Because it cannot expand abbreviations, the results may still not be as pleasing as the FP names, but this is inevitable if you have duplicate FP item names you do not want to rename individually.

Convert All Control Names to Lowercase--Converts control names to lowercase to conform to VXI *Plug&Play* standards.

Assign Instrument Driver Icon Based on Name--Assigns icons to VIs based on the name of the generated VIs. The converter searches for keywords such as initialize, close, self-test, reset, configure, or measure in the name of the function and uses the corresponding icon. If no keywords are found, a default icon is used. Additionally, the instrument prefix is stamped into the icon at the top left, up to a maximum of seven characters.

Default Size for Array Parameters--When an instrument driver DLL outputs an array, the memory into which the DLL writes the array must be preallocated and passed into the DLL. With this option, you can select the default size, in elements, to allocate for arrays. When a Call Library Node function

has an array as an argument, the converter drops an Initialize Array function to create an array to pass into the node. The initial size of such arrays is specified by **the Default Size for Array Parameters** option. A warning is generated in the .out file so that you can easily find VIs containing this construct and handle special cases individually.

Selecting **OK** at the main LabWindows/CVI Function Panel Converter dialog box brings up a file dialog box so you can select the library corresponding to the FP file being converted (if the **Create Library Call** option is set). Canceling this dialog box simply leaves the library path unspecified in the Call Library Function node; it does not abort conversion.

After you click **OK**, the converter brings up a working status dialog box to display the name of each new VI as it is created. A log file named `prefix.out` is created, listing all the VIs that were created, and any warnings or errors that occurred during conversion. If any warnings or errors do occur, you are notified to look at this file when conversion is complete.

Void Data Types

The type `void` is only allowed for the return value. This option is not available for parameters. Use it for the return value if your function does not return any values.

Application Management

This topic contains information about managing your files in LabVIEW.

File Arranging Using VI Libraries
Backing Up Your Files
VI Distribution
Multiple Developers
Creating Your Own Help Files

File Arranging Using VI Libraries

You can group multiple VI files by saving them as VI libraries (by convention, the names of these files end with the `extension.llb`). Or you can save VIs as individual files and group them within directories. For a comparison of the two methods, see the Saving VIs topic.

If you use VI libraries, you may want to divide your application into multiple VI libraries. Put the high level VIs in one VI library, and set up other VI libraries to contain VIs separated by function. It is easier to manage your VIs if you organize your files in this manner.

Another reason to break your VIs into multiple VI libraries is because saving files takes longer as the size of your VI library increases. When you save a VI into a VI library, the library is copied to the temporary directory, the VI is compressed and saved into the VI library, and the VI library is copied back to the original directory. This copying process helps prevent accidental corruption of the original file, but it also forces LabVIEW to take longer to save VIs.

A good rule of thumb is to avoid creating VI libraries that are over 1 MB. Notice there is no real limit on the size of a library, and the time to load VIs from VI libraries does not noticeably change regardless of the size of the VI library.

Another way to speed up the time it takes to save VIs, whether or not they are in VI libraries, is to make sure that your temporary directory is on the same drive (partition, in UNIX) as your VIs. LabVIEW copies files faster if the source and destination are on the same drive (or partition).

If you use VI libraries, you can emphasize the top level VIs by using **Edit VI Library...** from the File menu to mark certain VIs as Top Level. LabVIEW lists these files at the beginning of the file list in the File Dialog box. There is no similar option for directories, but you can emphasize your top level VIs by placing subVIs into subdirectories.

Backing Up Your Files

Accidents happen. Hard drives crash, and file systems can become corrupt. It is best to make a backup copy of your VIs at least once a week, and preferably once a day. You should copy them to another drive or to tape.

As an example of the problems that can occur, a LabVIEW user accidentally corrupted the VI library that contained several weeks of work. When prompted for a destination for a data file, this user had selected his VI library, at which point he wrote data over his own VI library. Fortunately he was able to recover most of the contents of his VI library, but only by good luck was it possible.

VI Distribution

When you are ready to distribute your VIs, consider whether you want to distribute the VIs with or without the block diagrams. Using the Save with Options dialog box, you can remove diagrams from your VIs. If a VI is saved without a diagram, it cannot be modified by users. If you save your VIs without diagrams, be careful not to overwrite your original versions.

Note: VIs without diagrams cannot be converted to future versions of LabVIEW or to other platforms. During conversion, LabVIEW has to recompile your VIs. Without a block diagram, a VI cannot be recompiled.

You should also consider what environment users will use to run your VIs. Do you want end users to have a development system or a run-time application?

A run-time application is appropriate when you do not expect your users to make modifications to your VIs. A run-time application contains simplified menus, and users cannot edit VIs or view block diagrams.

Run-time applications are built using the LabVIEW Application Builder libraries. When you build a run-time application with these libraries, you must answer these three questions.

- Do I want to embed a VI library in the application?
- Do I want the application to have an **Open** menu item?
- Do I want the application to have a **Quit** menu item?

If you choose to embed a VI library in the application, the library is combined with a LabVIEW run-time engine into a single file. When this file is launched, it automatically opens all top-level VIs in the library. If you do not embed a VI library, the application can be used to open any VI when it is launched, assuming the VI was saved with a development system for that platform.

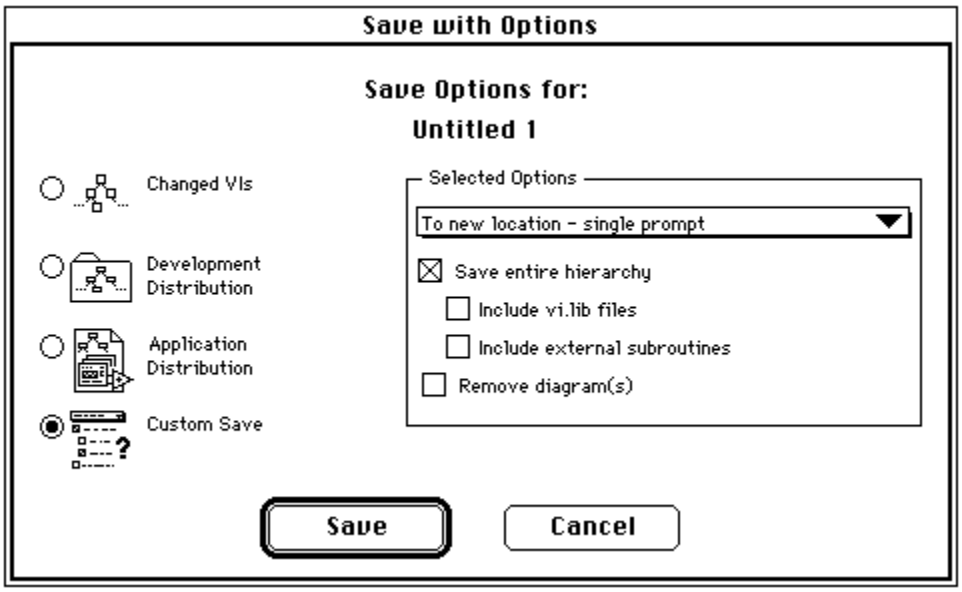
If you enable the **Open** menu item, the application can be used to open and run any VI in the file system, regardless of whether the application has an embedded VI library. By embedding a VI library, you can create a completely stand-alone application, one that prevents the user, or customer, from accessing the source VIs even if the user has the development system.

If you plan to ship multiple VI suites to the same customer, the single run-time application with the Open menu item enabled proves more efficient than separate embedded applications. This is true because each embedded application contains a copy of the run-time code, which is roughly 2 to 3 MB.

To create a run-time application, you need the Application Builder libraries, which are sold separately. See the LabVIEW Release Notes included with the Application Builder software for details on how to build an application.

Using Save with Options

The Save with Options dialog box makes it simple for you to save an entire hierarchy of VIs for distribution. You can use this dialog box to selectively save an entire hierarchy, without the VIs in `vi.lib`, and save all external subroutines that are referenced by VIs in your hierarchy. The Save with Options dialog box is shown in the following illustration.



By selecting from the options on the left of the Save with Options dialog box, you can select from a set of predefined save options. As you select options, the area at the right shows the behavior of your current selection. You can customize the behavior by selecting specific options from the right section of the dialog box.

The **Changed VIs** option saves any changes to the frontmost VI and its subVIs. This option is useful as a quick way to save changes.

The **Development Distribution** option saves all non-`vi.lib` VIs, controls, and external subroutines to a single location (either a directory or library). This option is an easy way to save a hierarchy that is to be transported to another LabVIEW development system, because the other development system has its own `vi.lib`.

The **Application Distribution** option saves all VIs, controls, and external subroutines, including the ones in `vi.lib` to a single location and removes the diagrams from all of the VIs. If you want to create a run-time application that has an embedded library, you can use this option to create the library to embed. To actually create a run-time application, you need the Application Builder libraries, which are sold separately.

You use the **Custom Save** option to pick the specific options you want from the **Selected Options** area of the dialog box, if none of the predefined options fit your needs.

Multiple Developers

If you have multiple developers working on the same project, you need to spend some design time defining responsibilities to ensure that the project works well.

Start by considering the top-level design of your application and create an outline. You should create stub VIs for the major components of your application. A stub VI is a prototype of a subVI. It has inputs and outputs, but is incomplete. It serves as a place holder for future VI development at which time you add functionality. Creating stub VIs is described in Chapter 10, *Program Design*, in your *LabVIEW Tutorial Manual*.

Once you have verified that the stub VIs provide the functionality you expect, you can determine which components individual developers should work on. To simplify matters, you might place the major stub VIs in separate directories or VI libraries which can then be managed by different developers.

Keeping a Master Copy
VI History Option
History Window

Keeping a Master Copy

You should probably keep the master copies of your project VIs on a single computer. You can institute a check-in/check-out policy to ensure control over what changes in your VIs. With this policy, a developer would check out a copy of the VIs, like checking a book out of the library. During that time, nobody should touch the files that the developer has checked out. A developer would check in the finished VIs after making any changes.

- There are several applications that can help control access to files and maintain different revisions of files. There is currently no way of performing a comparison between VIs, other than a visual comparison, so you need to be meticulous with your check-in/check-out policy to avoid multiple developers working on the same VIs at the same time.

VI History Option

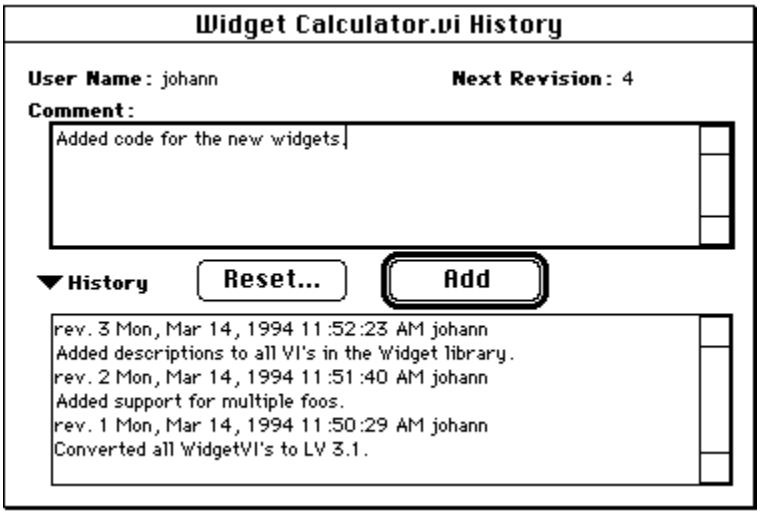
The VI History option is a feature intended to help developers keep track of changes to a VI as they are made. It does not provide a method for comparing two VIs to detect differences. Instead, it gives developers a place to record changes, and it assigns revision numbers to VIs.

History Window

In addition to its front panel and block diagram, a VI has a History window that displays the development history of the VI. Each developer who changes a VI can record any changes made in the history. The history is saved as part of the VI. It is made up of a series of comments entered by those who have made changes to the VI.

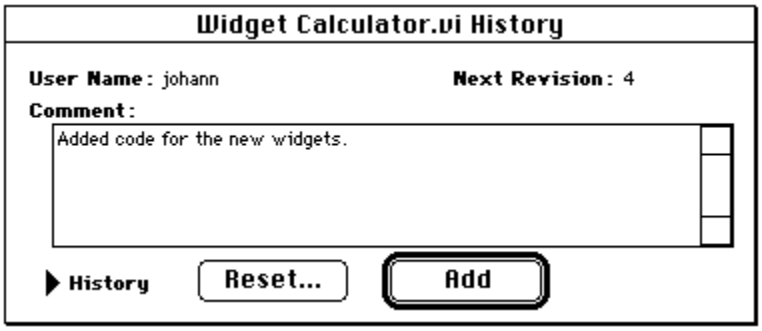
Note: Unless you have selected the Record comments generated by LabVIEW option in either the VI Setup»Documentation or the Preferences»History dialog box, the comments in a VI history are not automatically created. Anyone who makes changes to the VI must type in the information and keep the history up to date.

The History window is available whenever you are editing the VI by selecting **Windows»Show History** or pressing the key equivalent, <Ctrl-y> (Windows); <command-y> (Macintosh); <meta-y> (Sun); or <Alt-y> (HP-UX). The following illustration shows an example History window.



As you edit the VI, you can type a description of important changes into the Comment box near the top of the window. When you are finished making changes, you add the comment to the history by clicking on the **Add** button. If there is a comment in the Comment box when you save the VI, it is added to the history automatically. Once you add a comment, it is a permanent part of the history. You cannot rewrite the

history, so be sure to check your comments before adding them.
The box at the bottom of the window displays the VI history. LabVIEW shows a header for each comment in the history that includes the revision number of the VI, the date and time, and the name of the person who made the change. You can see that three comments have been added to this history, and each is displayed with a header. Because the preceding illustration is only an example, the comments are short; you can make them as long as you want when editing your own VIs.
The History window is fairly large with the history showing, but you can hide the history by clicking on the **History** button (the small black triangle). This makes the window small enough that it stays out of your way while you are editing the VI. When you want to see the history again, click on the same button and the history reappears. An example is shown in the following illustration.



You can also change the size of the history and comment boxes by resizing the window.
Revision Numbers
Resetting History Information
Printing History Information
Related VI Setup & Preference Dialog Box Options

Revision Numbers

The revision number is also a part of the history of the VI and is an easy way to see if the VI has changed (and how it changed, if you commented on your changes). The revision number starts at zero and is incremented every time you save the VI. The current revision number (the one on disk) is displayed in the **Get Info...** dialog box. It is also displayed in the titlebar of the VI if the option to do so is checked in the dialog box of **Edit**»**Preferences**»**History**.

The number displayed in the History window or VI window is the next revision number; that is, the number that is saved on disk if you save the VI. It is the current revision number plus one. When you add a comment to the history, the next revision number is included in the header of the comment. For example, if you were to click on the **Add** button in the example, your comment would be added to the history with the revision number 4. Then if you save the VI, the current revision number would be 4, and the comment that applies to the changes you just made would be labeled with the same number.

Note: The revision number will not be incremented when you save the VI if the only changes you made were changes to the VI History.

In many cases, there are gaps in revision numbers between comments because you saved the VI without entering a comment. This is not a problem. In fact, the revision number is useful because it is independent. Suppose you get a copy of a VI from another developer. Later you want to know if the other developer updated the VI since you got your copy. You can easily tell by looking at the revision numbers, even if the other developer made a change without adding comments.

If the revision number only changed when you added a comment, it would be a comment number, not a revision number. The revision number is even more effective when combined with the history. If developers have been adding comments to the history, you can tell what changes were made since you last got a copy.

Resetting History Information

Under the comment box there is a button labeled **Reset**. Pressing **Reset** erases the history and optionally reset the revision number to zero. This is useful when you copy a VI and want to start the new VI with a clean slate (no history).

Because the history is strictly a developer tool, the history is removed automatically when you remove the block diagram of a VI. The History window is not available in the run-time version of LabVIEW, but the revision number is available in the VI Information dialog box even for VIs that have had their block diagrams removed. You can remove the revision number using the **Reset** button to reset the history of your VI before you remove the block diagram.

Printing History Information

When you print a VI, you can include the history and revision number by selecting **Complete Documentation** from the Print Documentation dialog box. If you want to change the format of the printout to include only the history information, you can use the **Custom Print Settings** button to specify exactly what you want to print. Additionally, you can export the history information to a file by selecting **Save Text Info** from the Print Documentation dialog box.

Related VI Setup & Preference Dialog Box Options

The VI Setup and the Preferences dialog boxes both have options you can use to specify how VI history should behave.

You can use the VI Setup options to specify, for a given VI, whether history information is automatically recorded, and when. See the Setting Documentation Options topic for information on these settings.

You can use the Preferences dialog box options to specify what the default history settings are for new VIs. You can also use this dialog box to specify a user name entered in history information. See History Preferences for information on these settings.

Creating Your Own Help Files

You can create your own online help or reference documents if you have the right development tools. Help documents are based on formatted text documents. Among other things, you enter topics in these documents for your VIs to connect to.

The source files for all platforms must be in Windows Help format. It is possible to create help documents for multiple platforms.

Once you have created source documents, you use a help compiler to create a help document. If you want help files on multiple platforms, you must use the help compiler for the specific platform where they will be used. You may want to use any of the following compilers. The Windows compilers also include tools for creating help documents.

- ***(Windows)*** RoboHelp from Blue Sky Software, 1-800-677-4946; for international customers (619) 459-6365
- ***(Windows)*** Doc-To-Help from WexTech Systems, Inc., 1-800-939-8324.
- ***(Macintosh)*** QuickHelp from Altura Software, (408) 655-8005.
- ***(UNIX)*** HyperHelp from Bristol Technologies, (203) 438-6969

Once you have created and compiled your help files, you can link them directly to a LabVIEW VI. Pop up on the VI connector pane of the VI for which you want to link a file and select **VI Setup**»**Documentation**. Select the **Help Tag** box and type the topic you would like to link to in the help file. Choose the help file by clicking the **Browse...** button. The path of the file appears in the **Help Path** box.

Understanding How LabVIEW Executes VIs

This topic explains how LabVIEW multitasks.

Click here for overview material on multitasking.

With LabVIEW, you can run multiple VIs simultaneously. In addition, within a given VI you may have several parallel branches, each of which can also execute simultaneously.

In normal LabVIEW use, you do not need to be concerned with the details of how LabVIEW multitasks VIs. You can think of portions of a diagram as executing in parallel, and multiple VIs as running in parallel. With the LabVIEW multitasking capability, you can edit a VI while others run, or you can single step through a VI while others run at full speed. Also, if you inadvertently build a VI with an infinite loop, it should not lock up the computer or prevent other VIs from executing.

In some applications, such as process control applications, you may need a better understanding of how the LabVIEW multitasking system works.

Cooperative Multitasking
Nodes That Are Synchronous
Prioritizing Tasks of the Same Priority Level
Wait Functions to PrioritizeTasks
VI Setup Priority Setting
Reentrant Execution
Reentrant Execution Examples

Cooperative Multitasking

When LabVIEW compiles a diagram into machine code, the code is generated so that it periodically checks to see whether there are any other pending tasks and if it is time to let other tasks execute.

LabVIEW maintains a queue of active tasks. Assuming all tasks have the same priority, the first task executes for a certain amount of time. Then, that task is put at the end of the queue, and the next task executes for a time. When a task is finished, it is removed from the queue.

The execution system executes the first element of the queue by calling the generated code of the VI. At some point, the generated code of that VI checks with the execution system to see if it should allow another task to execute. If not, the VI's code continues to execute.

Task Switch Timing Occurence

The currently executing task surrenders execution to other tasks in a number of situations described in the following list.

1. The code of a VI occasionally checks to see how long it has been executing. If the VI has not had a complete amount of time (roughly 110 msecs), then it continues to execute. Otherwise, it surrenders to the scheduler.
2. If the VI encounters a Wait, Wait for Occurrence Dialog Box Function, or Device I/O function (which are used by the GPIB and Serial Port VIs), the VI surrenders to the scheduler.
3. When a VI surrenders to the scheduler, the scheduler selects the highest priority VI on its queue to run next. This can be the same VI that was running previously. The scheduler also may let the user interface code of LabVIEW handle mouse and key events before rerunning a VI.
4. If there is nothing of equal or higher priority, the VI continues to execute.

When a VI wait completes, or the I/O finishes, the VI is put back on the queue in order of its priority.

Nodes That Are Synchronous?

As mentioned previously, a few nodes are synchronous, meaning they do not multitask with other nodes. Code Interface Nodes (CINs) and all computation functions execute synchronously. Most analysis VIs contain CINs and therefore execute synchronously. For example, an FFT executes to completion without allowing other tasks to execute in parallel, regardless of how long the FFT executes.

Almost all other nodes are asynchronous. Structures, I/O functions, timing functions, and subVIs that you build all execute asynchronously.

Prioritizing Tasks of the Same Priority Level

There are two methods for prioritizing parallel tasks. One is to change a VI's priority using the priority setting of VI Setup. The other is to make strategic use of Wait functions.

In most cases, you probably should not change the priority of a VI from the default priority. Using priorities to control execution order may not give you the results you expect. If used incorrectly, using priorities to control execution order can result in completely pushing aside the lower priority tasks.

Wait Functions to PrioritizeTasks

Instead of using the priority setting in VI Setup, you can use the Wait function to make less important tasks execute less frequently. For example, if you have several loops in parallel and you want some to execute more frequently, put the Wait functions in the lower priority tasks. This causes them to execute less frequently, giving more time to other tasks.

In many cases, doing this is sufficient. You probably do not need to change the priorities using the VI Setup options.

VI Setup Priority Setting

If you decide to use priorities, and you do not want high priority VIs to push aside lower priority VIs, make sure to put Wait functions in lower-priority sections of the high priority VIs.

You change the priority of a VI using the **Priority** option in the VI Setup dialog box. There are five levels of priority: 0, 1, 2, 3, and subroutine, with 0 the lowest and subroutine the highest priority.

Higher priority VIs execute before lower priority VIs. That is, if the execution queue contains two VIs of each priority level, the level 3 VIs share execution time exclusively until both of them finish. Then, the level 2 VIs share execution time exclusively until both of them finish, and so on.

The exception to this occurs if the higher priority VIs call a Wait function or a Device I/O function (which are used by GPIB and Serial Port VIs). In this case, the higher priority VIs are taken off the queue until the Wait or I/O is complete, allowing other tasks (possibly with lower priority) to execute. Notice that as soon as the wait or I/O is complete, the pending task is reinserted on the queue in front of lower priority tasks.

Subroutine Priority Level

The previous discussion explains how multitasking works with priority levels 0 through 3. If you set a VI to subroutine priority (the highest level), the behavior is slightly different. When a VI runs at subroutine priority, no other VI can run until the subroutine priority VI is finished, even if the other VI is also marked as a subroutine. With subroutine VIs, LabVIEW does not multitask execution of VIs. In addition, subroutine VI execution is streamlined so that front panel controls and indicators are not updated when the subroutine is called. Watching a subroutine VI front panel reveals nothing about its execution.

A subroutine VI may call other subroutine VIs, but it may not call a VI with any other priority. Use

subroutine VIs in situations in which you want to put a simple computation having no interactive elements into a reusable subVI, where the call overhead for the subVI is minimized.

Reentrant Execution

Under normal circumstances, LabVIEW cannot execute multiple calls to the same subVI simultaneously. If you try to call a non-reentrant subVI from more than one place, one call executes and the other call waits for the first to finish before executing. If you make a VI reentrant (using VI Setup), each instance of the call maintains its own state of information. LabVIEW can then execute the same subVI simultaneously from multiple places. Reentrant execution is useful in the following situations.

- When a VI waits for a specified length of time or until a timeout occurs.
- When a VI contains data that is *not* to be shared between multiple instances of the same VI, as opposed to a global variable, which is a VI whose data you want to share.

You enable reentrant execution through the Execution Options of the VI Setup dialog box. If you select reentrancy, several other options become unavailable, including the following.

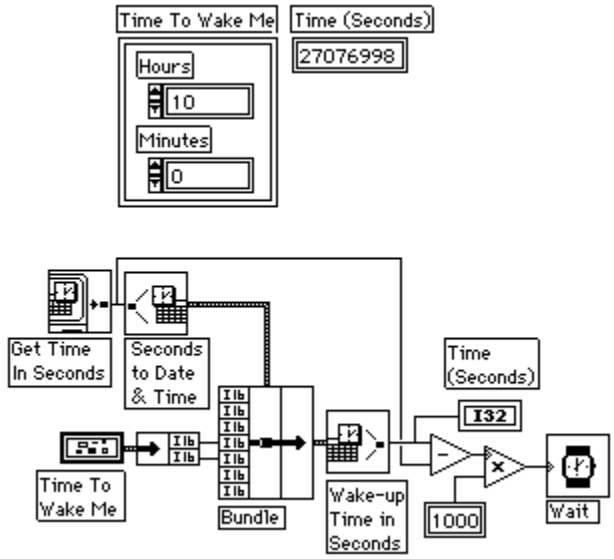
- Opening a VI or subVI when loaded
- Opening a subVI when called
- Execution highlighting
- Single-stepping

These options are disabled because the subVI must switch between different copies of the data and different execution states with each call, making it impossible to display its current state continuously.

Reentrant Execution Examples

Waiting VI

The following reentrant example discusses a VI, called Snooze, which takes hours and minutes as input and waits until that time arrives. If you want to use this simultaneously in more than one location, the VI needs to be reentrant.

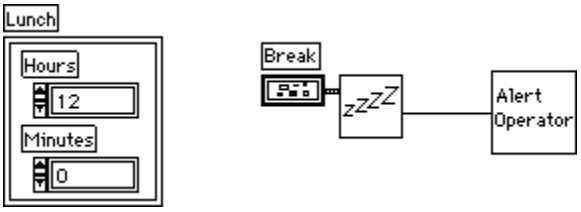


The Get Time In Seconds function reads the current time in seconds and the Seconds to Date/Time and converts this value to a cluster of time values (year, month, day, hour, minute, second, and day of week).

A Bundle function replaces the current hour and minute with values representing a later time on the same day in the front panel Time To Wake Me cluster control. The adjusted record is then converted back to seconds, and the difference between the current time in seconds and the future time is multiplied by 1,000 to obtain milliseconds. The result passes to a Wait function.

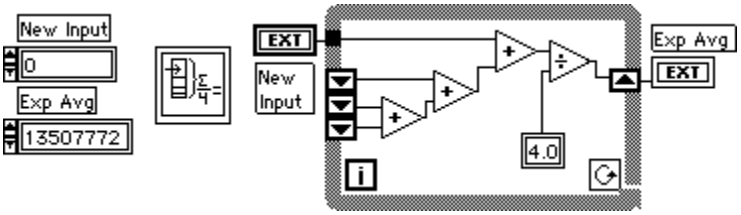
Lunch and Break are two VIs that use Snooze as a subVI. The Lunch VI, whose front panel and block diagram are shown in the following illustration, waits until noon and pops up a panel reminding the operator to go to lunch. The Break VI pops up a panel reminding the operator to go on break at 10:00 a.m. The Break VI is identical to the Lunch VI, except that the pop-up subVIs are different.

For Lunch and Break to execute in parallel, Snooze must be reentrant. Otherwise, if you started Lunch first, Break would have to wait until Snooze woke up at noon, which would be two hours late.

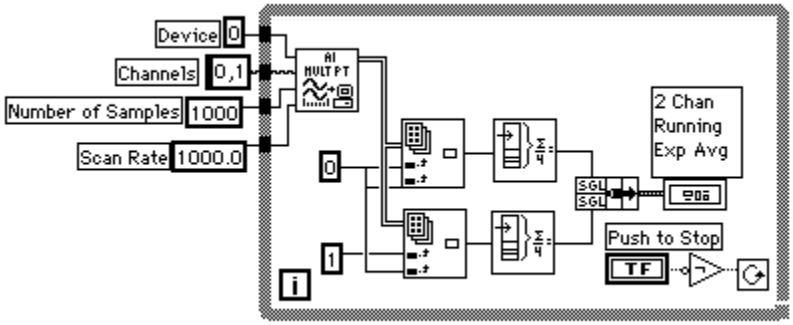


Storage VI Not Meant to Share Its Data

Another situation that requires reentrancy is when you make multiple calls to a subVI that stores data. Suppose you create a subVI, ExpAvg, which calculates a running exponential average of four data points. To remember the past values, ExpAvg uses an uninitialized shift register with three left terminals. See Chapter 3, *Loops and Charts*, of *your LabVIEW Tutorial Manual* for more information on uninitialized shift registers.



Next, suppose you have a VI that uses ExpAvg to calculate the running average of two data acquisition channels. For example, you are monitoring the voltages at two points in a process and want to view the exponential running average on a strip chart. The diagram contains two ExpAvg nodes. The calls alternate, one for Channel 0, then one for Channel 1. Assume Channel 0 executes first. If ExpAvg is not reentrant, the call for Channel 1 uses the average computed by the call for Channel 0, and vice versa. By making ExpAvg reentrant, each call can execute independently without danger of sharing the data.



Multitasking Overview

Most computers have only one processor, meaning only one task can execute at any given time. Multitasking is achieved by running one task for a short amount of time and then letting other tasks run. As long as the amount of time that each task gets is small enough, you get the appearance of having multiple tasks running simultaneously.

There are two kinds of multitasking--preemptive and cooperative. With a preemptive multitasking system, each task is given a limited amount of time to execute. When the time is up for a given task, one task is forced to pause while another starts execution. This enforcement is handled by either the operating system or by the execution system that is controlling the tasks. Preemptive multitasking is not currently available on all operating systems and is not currently used by LabVIEW.

Cooperative multitasking, which is more common in current, small computer operating systems, relies on each task being written to share time. Each task is responsible for deciding when it should give up time to other tasks. If any task fails to share time, that task ends up locking out other tasks until it is ready to share time.

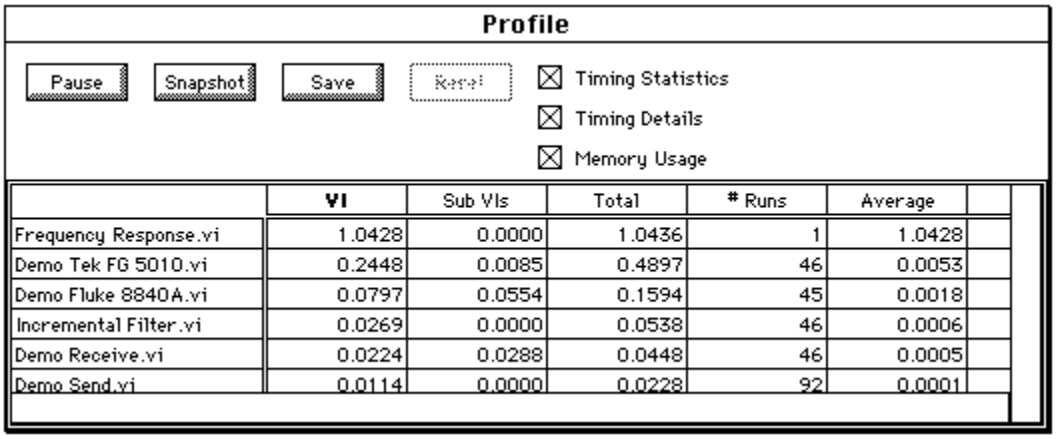
LabVIEWs execution system is a cooperative multitasking system, meaning that LabVIEW can execute multiple tasks in parallel. Because it is cooperative, you have no assurances that a given task executes within a specific amount of time. Some nodes in LabVIEW are synchronous, meaning they do not multitask (a complete list of these nodes is given in the Nodes That Are Synchronous topic). If any synchronous nodes execute a lengthy process, other tasks may be temporarily locked out of execution. In practice, this rarely causes problems because the number of synchronous nodes is relatively limited, and the actions they execute generally take only a small amount of time.

Performance Profiling

The VI Performance Profiler is a powerful tool for determining where your application is spending its time, as well as how it is using memory. This information is invaluable in finding the hot-spots of your application so that you can optimize the VIs that take up the most time. It provides an interactive tabular display of time and memory usage for each VI in your system. Each row of the table contains information for a specific VI. The time spent by each VI is broken down into several categories as well as summarized in a few statistics. The memory usage is broken down into minimum, maximum, and average per run of a VI.

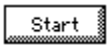
The interactive tabular display allows you to view all or parts of this information, sort it by different categories, and look at the performance data of subVIs when called from a specific VI.

Select **Project»Show Profile Window** to bring up the Profile window. The following illustration shows an example of the window already in use.

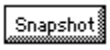


VI	Sub VIs	Total	# Runs	Average
Frequency Response.vi	1.0428	0.0000	1.0436	1	1.0428
Demo Tek FG 5010.vi	0.2448	0.0085	0.4897	46	0.0053
Demo Fluke 8840A.vi	0.0797	0.0554	0.1594	45	0.0018
Incremental Filter.vi	0.0269	0.0000	0.0538	46	0.0006
Demo Receive.vi	0.0224	0.0288	0.0448	46	0.0005
Demo Send.vi	0.0114	0.0000	0.0228	92	0.0001

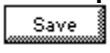
There are several things to notice about the window. First, the collection of memory usage information is optional. This is because the collection process can add a significant amount of overhead to the running time of your VIs. You must choose whether to collect this data before starting the Profiler by checking the **Profile Memory Usage** checkbox appropriately. This checkbox can not be changed once a profiling session is in progress.



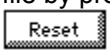
To enable the collection of performance data, you must press the **Start** button in the Profile window. It is best to start a profiling session while your application is not running. This way you can ensure that you measure only complete runs of VIs, and not partial runs.



While the Profiler is enabled, you can view the data that is currently available by pressing the **Snapshot** button. This gathers data from all the VIs in memory and displays it in the tabular display.



You can also save the currently displayed data to disk as a tab-delimited text spreadsheet file by pressing the **Save** button. This data can then be viewed in a spreadsheet program or by VIs.



If you wish to erase all the current Profiler information from the table so that you can start a new profiling session, press the **Reset** button.

Performance Profile Results
Timing Information
Memory Information

Performance Profile Results

You can choose to display only parts of the information in the table. Some basic data is always visible, but you can optionally display the statistics, details, and (if enabled) memory usage by checking or unchecking the appropriate checkboxes in the Profile window.

Notice that performance information is also displayed for Global VIs. However, this information sometimes requires a slightly different interpretation, as described in the category-specific sections below.

Performance data for subVIs when called from a specific VI can be viewed by double-clicking on a VI's name in the tabular display. When you do this, new rows appear directly below the VI'sname containing performance data for each of its subVIs. When you double-click on the name of a Global VI, new rows appear for each of the individual controls on its front panel.

You can sort the rows of data in the tabular display by clicking in the desired column header. The current sort column is indicated by a bold header title.

Timings of VIs may not necessarily correspond to the amount of elapsed time that it takes for a VI to complete. This is because LabVIEW's multi-threaded execution system may interleave the execution of two or more VIs. Also, there is a certain amount of overhead taken up by LabVIEW that is not attributed to any VI, such as the amount of time taken by a user to respond to a dialog box, or time spent in a Wait function on a block diagram, or time spent by LabVIEW to check for mouse clicks.

The basic information that is always visible in the first three columns of the performance data consists of the following items:

- VI Time--The total time spent by this VI. This is the time spent actually executing this VI's code and displaying its data, as well as time spent by the user interacting with its front panel controls (if any). This summary is broken down into sub-categories in the **Timing Details** described below. For Global VIs, this time is the total amount of time spent copying data to or from all of its controls. To get timing information on individual controls in a Global VI, you can double-click the Global VI's name.
- SubVIs' Time--The total time spent by all subVIs of this VI. This is the sum of the VI time (described above) for all callees of this VI, as well as their callees, etc.
- Total Time--This is the sum of the above two categories, giving the total amount of time.

Timing Information

When the **Timing Statistics** checkbox is checked, the following columns become visible in the tabular display:

- # Runs--The number of times that this VI completed a run. For Global VIs, this time is the total number of times all of its controls were accessed.
- Average--The average amount of time spent by this VI per run. This is simply the VI time divided by the number of runs.
- Shortest--This is the minimum amount of time the VI spent in a run.
- Longest--This is the maximum amount of time the VI spent in a run.

When the **Timing Details** checkbox is checked, you can view a breakdown of several timing categories that sum up to the time spent by the VI. For VIs that have a lot of user interface, these categories can help you see what operations are taking the most time.

- Diagram--This is the time spent only executing the code that was generated for the diagram of this VI.
- Display--This is the time spent updating front panel controls of this VI with new values from the diagram.
- Draw--This is the time LabVIEW spent drawing the front panel of this VI. Draw time includes the time it takes to simply draw a front panel when its window has just been opened, or when it is being revealed after being obscured by another window. Draw time also includes time that is conceptually Display time, because it is caused by new values coming in from the diagram, but occurs because the control is transparent and/or overlapped. These controls must invalidate their area of the screen when they receive new data from the diagram so that everything in that area can redraw in the correct

order. Other controls can draw immediately on the front panel when they receive new data from the diagram. More LabVIEW overhead is involved in invalidating and redrawing--most (but not all) of which shows up in the Draw timings.

- Tracking--This is the time LabVIEW spent tracking the mouse while the user was interacting with the front panel of this VI. This can be significant for some kinds of operations, such as zooming in or out of a graph, selecting items from a pop-up menu, or selecting and typing text in a control.
- Locals--This is the time spent copying data to or from local variables on the diagram. Experience with users has shown that this time can sometimes be significant, especially when it involves large, complex data.

Memory Information

When the **Memory Usage** checkbox is checked (remember that this is only available if the **Profile Memory Usage** checkbox was selected before you began the profiling session), you can view information about how your VIs are using memory. These values are a measure of the memory used by the VI's data space and do not include the LabVIEW support data structures necessary for all VIs. The VI's data space contains not just the data explicitly being used by front panel controls, but also temporary buffers that are implicitly created by the LabVIEW compiler.

The memory sizes are measured at the conclusion of a VI's run and may not exactly reflect its total usage. For instance, if a VI creates large arrays during its run but reduces their size before the VI finishes, the sizes displayed will not reflect the intermediate larger sizes.

Two sets of data are displayed in this topic: that related to the number of bytes used and that related to the number of blocks used. Independent blocks of memory are used for arrays, strings, paths, and pictures (from the Picture Control Kit). Large numbers of blocks in your application's memory heap can cause an overall degradation of performance throughout LabVIEW (not just execution).

The categories of Memory Usage are the following:

- Average Bytes--The average number of bytes used by this VI's data space per run.
- Min Bytes--The minimum number of bytes used by this VI's data space for an individual run.
- Max Bytes--The maximum number of bytes used by this VI's data space for an individual run.
- Average Blocks--The average number of blocks used by this VI's data space per run.
- Min Blocks--The minimum number of blocks used by this VI's data space for an individual run.
- Max Blocks--The maximum number of blocks used by this VI's data space for an individual run.

Speeding Up Your VIs

LabVIEW compiles your VIs and produces code that generally executes very quickly. When working on time critical applications, you may want to do all you can to get the best performance out of your VIs. This section discusses factors that affect execution speed and suggests some programming techniques to help you get the best performance possible.

You should examine the following items to determine the causes of slow performance.

- Input/Output (files, GPIB, data acquisition, networking)
- Screen Display (large controls, overlapping controls, too many displays)
- Memory Management issues (inefficient usage of arrays and strings, inefficient data structures)

Other factors, such as execution overhead and subVI call overhead can have an affect, but these are usually minimal and not the most critical source of slow execution.

Input/Output

I/O calls generally incur a lot of overhead. They often take an order of magnitude more time than the time it takes to perform a computational operation. For example, a simple serial port read operation may have an associated overhead of several milliseconds. This amount of overhead is true not only for LabVIEW, but also for any other application. The reason for this overhead is that an I/O call involves transferring information through several layers of an operating system.

The best method for addressing too much overhead is to minimize the number of I/O calls you make. Your performance improves if you can structure your application so that you transfer a lot of data with each call, instead of making multiple I/O calls using smaller amounts of data.

For example, if you are creating a data acquisition (DAQ) VI, you have a couple of options for reading data. You can use a single-point data transfer function such as the AI Sample Channel VI or you can use a multi-point data transfer function such as the AI Acquire Waveform VI. If you need to acquire 100 points, you could use the AI Sample Channel VI in a loop with a Wait function to establish the timing. Or you can use the AI Acquire Waveform VI with an input specifying that you want 100 points.

You can get much higher and more accurate data transfer rates by using the AI Acquire Waveform VI, because it uses hardware timers to manage the data transfer. In addition, overhead for the AI Acquire Waveform VI is roughly the same as the overhead for a single call to the AI Sample Channel VI, even though it is transferring much more data.

Screen Display

Updating controls on a front panel can frequently be one of the most time expensive operations in an application. This is especially true if you use some of the more complicated displays, such as graphs and charts.

This overhead is minimized to a certain extent by the fact that most of LabVIEW's controls are intelligent; they do not redraw when they receive new data if the new data is the same as the old data. Graphs and charts are exceptions to this rule.

If redraw rate becomes a problem, the best solutions are to reduce the number of controls you use, and keep the displays as simple as possible. In the case of graphs and charts, you can turn off autoscaling, scale markers, and grids to speed up displays.

If you have controls that are overlapped with other objects, their display rate is cut down significantly. The

reason for this is that if a control is partially obscured, LabVIEW has to go through more work to redraw that area of the screen. Unless you have the Smooth Updates preference on, you will probably see more flicker when controls are overlapped.

As with other kinds of I/O, there is a certain amount of fixed overhead in the display of a control. You can pass multiple points to an indicator at one time using some controls, such as charts. You can minimize the number of chart updates you need to make by passing more data to the chart each time. You get much higher data display rates if you collect your chart data into arrays so that you can display multiple points at a time, instead of displaying each point as it comes in.

When you design subVIs whose panels are closed during execution, you do not need to worry about display overhead. If the panel is closed, you do not have the drawing overhead for controls, so graphs are no more expensive than arrays.

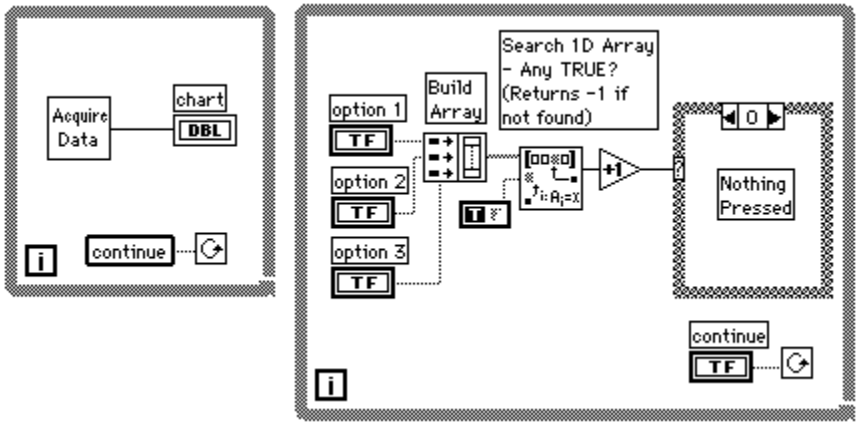
Memory Management Issues

Parallel Diagrams
SubVI Overhead
Unnecessary Computation in Loops

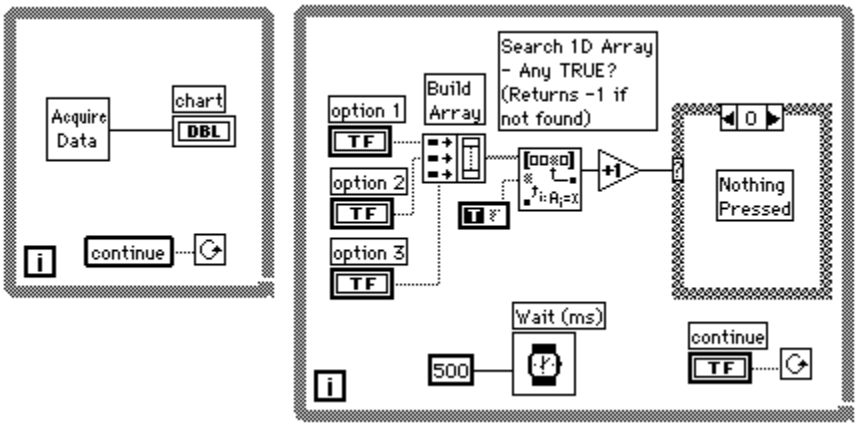
Parallel Diagrams

When you have multiple diagrams running in parallel, LabVIEW switches between them periodically. If some of these loops are less important than others, you should use the Wait function to ensure that the less important loops use less time.

For example, consider the following diagram.



There are two loops in parallel. One of the loops is acquiring data, and needs to execute as frequently as possible. The other loop is monitoring user input. Given the way this program is structured, the loops get equal time. The loop monitoring the user's action gets a chance to run several times a second.
In practice, it is probably okay if the loop monitoring the button executes only once every half second, or even less often. By calling the Wait (ms) function in the user interface loop, you give significantly more time to the other parallel loop.



SubVI Overhead

Whenever you call a subVI, there is a certain amount of overhead associated with the call. This overhead is fairly small (on the order of tens of microseconds), especially in comparison to I/O overhead and display overhead, which can range from milliseconds to tens of milliseconds.

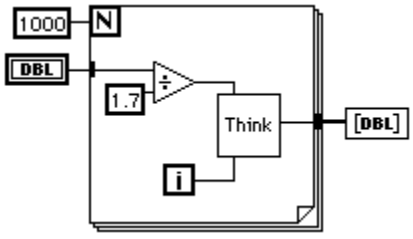
However, this overhead can add up in some cases. For example, if you call a subVI 10,000 times in a loop, this overhead may take up a significant amount of time. In this case, you may want to consider whether the loop could be embedded in the subVI.

Another option that you might consider is turning certain subVIs into subroutines (using the VI Setup Priority option). When a VI is marked as a subroutine, LabVIEW minimizes the overhead to call a subVI. There are a few trade-offs, however. Subroutines cannot display front panel data (LabVIEW does not copy data from or to subroutines' front panel controls), they cannot contain timing or dialog box functions, and they do not multitask with other VIs. Subroutines are generally most appropriate for VIs that do not require user interaction and are short, frequently executed tasks.

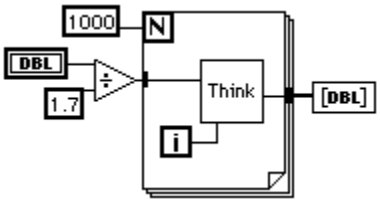
Unnecessary Computation in Loops

Avoid putting calculations in loops if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop.

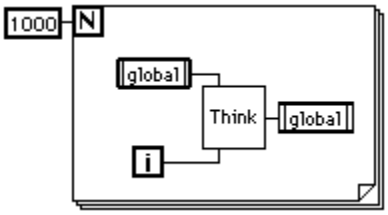
For example, consider the following diagram.



The result of the division is the same every time through the loop; therefore you can increase performance by moving the division out of the loop, as shown in the following illustration.

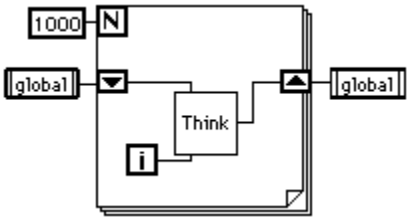


Now, consider the diagram in the illustration that follows.

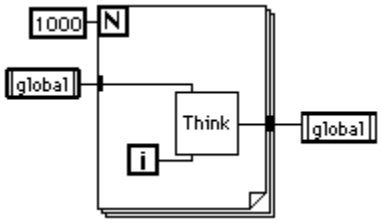


If you know that the value of the global variable is not going to be changed by another concurrent diagram or VI during this loop, this diagram wastes time by reading from the global and writing to the global every time through the loop.

If the global variable is not read from or written to by another diagram during this loop, you might use the following diagram instead.



Notice that you need the shift registers to pass the new value from the subVI to the next iteration of the loop. The following diagram is a common mistake that some beginning LabVIEW users make. Since there is no shift register, the results from the subVI are never passed back to the subVI as its new input value.



Memory Usage

Many of the details which you normally have to worry about in a conventional language, LabVIEW handles transparently, behind the scenes. One of the main challenges of a conventional language is memory usage. In a conventional language, you, the programmer, have to take care of allocating memory before you use it, and deallocating it when you are finished. You also have to be particularly careful not to accidentally write past the end of the memory you allocated in the first place. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes that programmers make in conventional text-based languages. It is also a difficult problem to debug.

LabVIEWs dataflow paradigm removes much of the difficulty of managing memory. In LabVIEW, you do not allocate variables, nor assign values to and from them. Instead, you create a diagram with connections representing the transition of data.

Functions that generate data take care of allocating the storage for that data. When data is no longer being used, LabVIEW may deallocate the associated memory. When you add new information to an array or a string, LabVIEW takes care of the memory allocation that is involved.

This automatic memory handling is one of the chief benefits of LabVIEW. However, because it is automatic, you have less control over when it happens. If your program works with large sets of data, it is important to have some understanding of when LabVIEW allocates and deallocates memory. An understanding of the principles involved can result in programs with significantly smaller memory requirements. Also, an understanding of how to minimize memory usage can also help to increase VI execution speeds because memory allocation and copying data can take a considerable amount of time.

Memory Usage Basic Concepts
Memory Usage Monitoring
General Rules for Better Memory Usage
Memory Usage Rules In Detail
Efficient Data Structures
Case Study 1: Avoiding Complicated Data Types
Case Study 2: Global Table of Mixed Data Types
Case Study 3: A Static Global Table of Strings

Memory Usage Basic Concepts

Memory Storage (Macintosh)
Virtual Memory (All Platforms)
VI Components
Dataflow Programming and Data Buffers

Memory Storage (Macintosh)

When LabVIEW for Macintosh is launched, the system allocates for it a single block of memory (called the heap), out of which all subsequent allocations are performed. When you load VIs, components of those VIs are loaded into the heap. Likewise, when you run a VI, all the memory that it manipulates is allocated out of the heap.

You configure the amount of memory that the system allocates at launch time using the Get Info command from the Finder. Notice that if LabVIEW runs out of memory, it cannot increase the size of this memory pool. Therefore, you should set up this parameter to be as large as is practical. If you have a 16 MB machine, consider the applications that you want to run in parallel with LabVIEW. If you do not plan to run any other applications, set the memory preference to be as large as possible.

Virtual Memory (All Platforms)

If you have a machine with a limited amount of memory, you may want to consider using virtual memory to increase the amount of memory available for applications. Virtual memory is a capability of your operating system by which it uses available disk space for RAM storage. If you allocate a large amount of virtual memory, applications perceive this as memory that is generally available for storage.

LabVIEW does not care whether your memory is real RAM or virtual memory. The operating system hides the fact that the memory is virtual. The main difference is speed. With virtual memory, you may occasionally notice more sluggish performance, when memory is swapped to and from the disk by the operating system. Virtual memory can help run larger applications, but it is probably not appropriate for applications that have critical time constraints.

VI Components

A VI has four major components.

- Front panel
- Block diagram
- Code (diagram compiled to machine code)
- Data (control and indicator values, default data, diagram constant data, and so on.)

When you load a VI, LabVIEW loads the front panel, the code (if it matches the platform), and the data for the VI into memory. If the VI needs to be recompiled because of a change in platform or a change in the interface to a subVI, then LabVIEW loads the diagram into memory as well.

LabVIEW also loads the code and data space of subVIs into memory. Under certain circumstances, LabVIEW loads the front panel of some subVIs into memory as well. This can occur, for example, if the subVI uses Attribute Nodes, because Attribute Nodes manipulate state information for front panel controls.

An important point in the organization of VI components is that you generally do not use too much memory when you convert a section of your VI into a subVI. If you create a single, huge VI with no subVIs, you end up with the front panel, code, and data for that top-level VI in memory. If the VI is broken into subVIs, the code for the top-level VI is smaller, and the code and data of the subVIs is in memory. In some cases, you may actually see lower run-time memory usage. This idea is discussed later in this chapter in the section, How to Monitor Memory Usage.

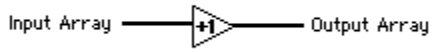
You may also find that massive VIs take longer to edit. You do not see this problem as much if you break your VI into subVIs, because the LabVIEW editor can handle smaller VIs more efficiently. This is in addition to the fact that a more hierarchical organization to your VIs is generally easier to maintain and read.

Note: If the front panel or block diagram of a given VI is much larger than a screen, you may want to break it into subVIs to make it more accessible.

Dataflow Programming and Data Buffers

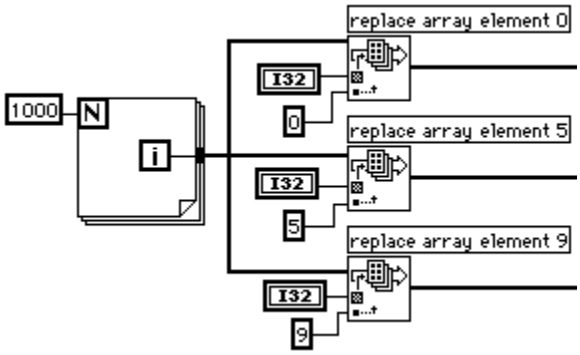
In dataflow programming, you do not generally use variables. Dataflow models usually describe nodes as consuming data inputs and producing data outputs. A literal implementation of this model would produce applications that can use very large amounts of memory and have sluggish performance. Every function would produce a copy of data for every destination to which an output is passed. LabVIEW improves on this implementation by attempting to determine when memory can be reused, and by looking at the destinations of an output to determine whether it is necessary to make copies for each individual terminal.

For example, if LabVIEW took a more traditional approach (as it did in early versions of LabVIEW 1), the following diagram would use two blocks of data memory, one for the input and one for the output.

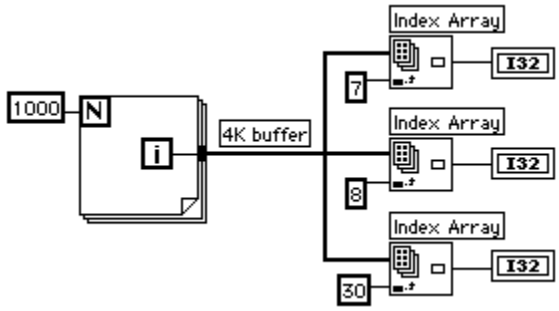


The input array and the output array contain the same number of elements, and the data type for both arrays is the same. Think of the incoming array as a buffer of data. Instead of creating a new buffer for the output, LabVIEW reuses the input buffer. This saves memory, and also results in faster execution, because no memory allocation needs to take place at run time.

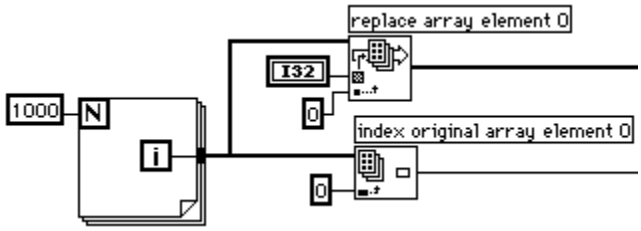
This reuse of memory buffers cannot take place in all cases. For example, consider the following diagram.



A single source of data is being passed to multiple destinations. The destinations want to modify the data to produce resulting arrays. In this case, LabVIEW creates new data buffers for two of the functions and copies the array data into the buffers. Thus, one of the functions is reused by the input array, and the others are not. This diagram uses about 12 KB (4 KB for the original array and 4 KB for each of the extra two data buffers).
Now, examine the following diagram.



As before, the input branches to three functions. However, in this case none of them need to modify the data. If you pass data to multiple locations, all of which read the data without modifying it, LabVIEW does not need to make a copy of the data. This diagram uses up about 4 KB of data.
Finally, consider the following diagram.



In this case, the input branches to two functions, one of which wants to modify the data. There is no dependency between the two functions. Therefore, you would expect that at least one copy would need to

be made so that the replace array element function could safely modify the data. In this case, LabVIEW schedules the execution of the functions in such a way that the function that wants to read the data executes first, and the function that wants to modify the data executes last. This way, the Replace Array Element function reuses the incoming array buffer without generating a duplicate array. If the ordering of the nodes is important, you should make the ordering explicit by using either a sequence or an output of one node for the input of another.
In practice, LabVIEW's analysis of diagrams is not perfect. In some cases, LabVIEW may not be able to determine the optimal method for reusing diagram memory.

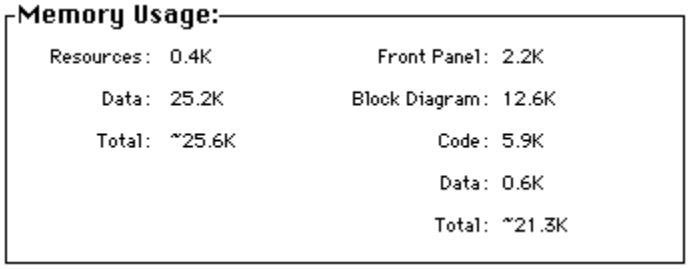
Memory Usage Monitoring

There are a couple of methods for determining memory usage.

If you select **About LabVIEW...**, you get statistics that summarize the total amount of memory that has been used. Notice that this memory includes memory for VIs as well as memory that LabVIEW uses. You can check this amount before and after execution of a set of VIs to get a rough idea of how much memory is being used.

You can get a view of the dynamic usage of memory by your VIs with the Performance Profiler. It keeps statistics on the minimum, maximum, and average number of byes and blocks used by each VI per run. See the Performance Profiling topic for more details.

As shown in the following illustration, you can use **Get Info...** to get a breakdown of the memory usage of a given VI. The left column of this information summarizes disk usage, and the right column summarizes how much RAM is currently being used for various components of the VI. Notice that these statistics do not include memory usage of subVIs.



A fourth method for determining memory usage is to use a VI called the Memory Monitor VI (in `memmon.llb` inside of the `Examples` directory). This VI uses CINs to find out memory usage for all VIs in memory.performance issues:memory usage:monitoring

General Rules for Better Memory Usage

The main point of the previous discussion is that LabVIEW attempts to reuse memory intelligently. The rules for when LabVIEW can reuse memory and when it cannot are fairly complex, and are discussed in the Memory Usage Rules In Detail topic. In practice, the following rules should help you to create VIs that use memory efficiently.

- Breaking a VI into subVIs generally does not hurt your memory usage, In fact, in many cases, you see an improvement, because LabVIEW can reclaim subVI data memory when the subVI is not executing.
- Do not worry too much about copies of scalar values; it takes a lot of scalars to make a dent in memory usage.
- Do not overuse global and local variables when working with arrays or strings; reading a global or local variable causes a copy of the data of the variable to be generated.
- On open panels, do not display large arrays and strings unless it is necessary. Indicators on open panels retain a copy of the data that they display.

- If the panel of a subVI is not going to be displayed, do not leave unused Attribute Nodes on the subVI. Attribute Nodes cause the panel of a subVI to remain in memory, which may cause more memory usage.
- Do not use suspend data range on time/memory critical VIs. The panel for the subVI needs to be loaded to check range checking, and extra copies of data are made for the subVIs controls and indicators.
- In designing diagrams, watch for places where the size of an input is different from the size of an output. For example, if you see a lot of places where you are increasing the size of an array or string frequently using Build Array or Concatenate Strings, you are generating copies of data.
- Use consistent data types for arrays and watch for coercion dots when passing data to subVIs and functions; whenever LabVIEW changes data types, the output is a new buffer.
- Do not use complicated, hierarchical data types (for example, arrays of clusters containing large arrays or strings, or clusters containing large arrays or strings). You may end up using more memory, and performance suffers. See the topic on Efficient Data Structures for more details and suggestions for tactics in designing your data types.

Memory Usage Rules In Detail

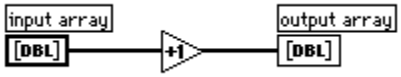
The following topics describe the reasoning behind the preceding rules in more detail.

Memory Usage in Front Panels
SubVIs Can Reuse Data Memory
Local Variables Cannot Reuse Data Memory
Global Variables Always Keep Copies of Their Data
When Does LabVIEW Deallocate Memory?
When Can an Output Reuse an Input Buffer?
Use Consistent Data Types

Memory Usage in Front Panels

When a panel is open, front panel controls and indicators keep their own, private copy of the data that they display.

Consider the following example of the increment function, with the addition of front panel controls and indicators.



When the VI is run, the data of the front panel control is passed to the diagram. The increment function reuses the input buffer. The indicator then makes a copy of the data for display purposes. Thus there are three copies of the buffer.
This data protection of the front panel control prevents the case where you enter some data into a control, run the associated VI, and see the data change in the control as it is passed in-place to subsequent nodes. Likewise, data is protected in the case of indicators so that they can reliably display the previous contents until they receive new data.
With subVIs, you can use controls and indicators as inputs and outputs. LabVIEW makes a copy of the control and indicator data of the subVI in the following conditions.

- The front panel is in memory--this can occur for any of the following reasons.
  - The front panel is open.
  - The VI has been changed, but has not been saved (all components of the VI remain in memory until the VI has been saved).
  - The panel uses data printing.

    - The diagram uses Attribute Nodes.
- The VI uses local variables.
- The panel uses data logging.
- A control uses suspend data range checking.

A few of these reasons are not intuitive and need further explanation.

Consider attributes such as chart history. For an Attribute Node to be able to read the chart history in subVIs that are not open, LabVIEW needs to go through the motions of displaying the data that is passed to controls and indicators. Because there are numerous other attributes like this, LabVIEW keeps subVI panels in memory if the subVI uses Attribute Nodes.

If a front panel uses front panel datalogging or data printing, then controls and indicators maintain copies of their data. In addition, panels are kept in memory for data printing so that the panel can be printed out.

If a VI uses Suspend data range checking, data is copied to and from all front panel controls and indicators. The front panel values are retained so that the front panel can be displayed if any data goes out of range. Do not use Suspend range checking if memory and speed are a major concern.

Notice that if you have set a subVI to display its front panel when called using VI Setup or SubVI Setup, the panel is loaded into memory when the subVI is called. If you have set **the Close if Originally Closed** option, the panel is removed from memory when the subVI finishes execution.

SubVIs Can Reuse Data Memory

In general, a subVI can use data buffers from its caller as easily as if the subVI's diagram were duplicated at the top level. For most subVIs, you do not use more memory if you convert a section of your diagram into a subVI. For VIs with special display requirements, as described in the previous section, there may be some additional memory usage for front panels and controls.

Local Variables Cannot Reuse Data Memory

When you create subVIs, you create a connector pane that describes how data is passed to and from the subVI. The data buffers that come from terminals that are connected to a connector pane can reuse data buffers from calling VIs. Local variables cannot do this. Instead, whenever you read from a local variable, you create a new buffer for the data from its associated control.

If you use local variables to transfer large amounts of data from one place on the diagram to another, you generally use more memory, and consequently have slower execution speed than if you can transfer data using an alternative wire path.

Global Variables Always Keep Copies of Their Data

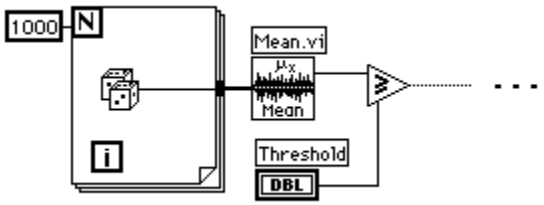
Whenever you read from a global variable, you create a copy of the data that is stored in that global variable.

When manipulating large arrays and strings, the time and memory needed to manipulate global variables can be considerable. This is especially inefficient when dealing with arrays if you only want to modify a single array element, then store the entire array. If you read from the global variable in several places in your diagram, you may end up creating several buffers of memory.

One technique for minimizing memory usage in this case is to use an uninitialized shift register in a subVI. This technique can provide the compactness of a global variable with the efficiency of a shift register.

When Does LabVIEW Deallocate Memory?

Consider the following diagram.



After the Mean VI has executed, the array of data is no longer needed. Because determining when data is no longer needed can become very complicated in larger diagrams, LabVIEW does not deallocate the data buffers of a particular VI during its execution.
If LabVIEW is low on memory, it essentially deallocates data buffers used by any VI that is not currently executing. LabVIEW does not deallocate memory used by front panel controls, indicators, global variables, or unitialized shift registers.
Now, consider the same VI described previously as a subVI to a larger VI. The array of data is created and used only in the subVI. If the subVI is not executing and LabVIEW is low on memory, it may deallocate the data in the subVI. This is a case in which using subVIs can actually save on memory usage.

When Can an Output Reuse an Input Buffer?

If an output is the same size and data type as an input, and the input is not needed elsewhere, the output can reuse the input buffer. As mentioned previously, in some cases even when an input is used elsewhere, LabVIEW can order code execution in such a way that it can reuse the input for an output buffer; however, the rules for this are complex and should not be counted on.

Use Consistent Data Types

If an input has a different data type from an output, the output cannot reuse that input. For example, if you add a 32-bit integer to a 16-bit integer, you see a coercion dot that indicates that the 16-bit integer is being converted to a 32-bit integer. The 32-bit integer input may be usable for the output buffer, assuming it meets all of the other requirements (for example, the 32-bit integer is not being reused somewhere else).

In addition, coercion dots for subVIs and many functions imply a conversion of data types. In general, LabVIEW creates a new buffer for the converted data.

To minimize memory usage, you should use consistent data types wherever possible. Doing this produces fewer copies of data because of promotion of data in size. Using consistent data types also gives LabVIEW more flexibility in determining when data buffers can be reused.

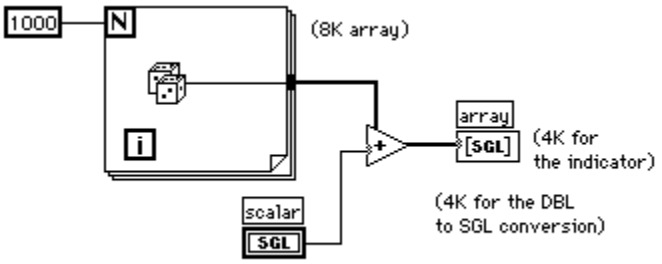
In some applications, you might consider using smaller data types. For example, you might consider using four-byte, single-precision numbers instead of eight-byte, double-precision numbers. However, you should carefully consider which data types are expected by subVIs you may call, because you want to avoid unnecessary conversions.

How to Generate Data of the Right Type
Avoid Constantly Resizing Data
Example 1: Building Arrays
Example 2: Searching through Strings

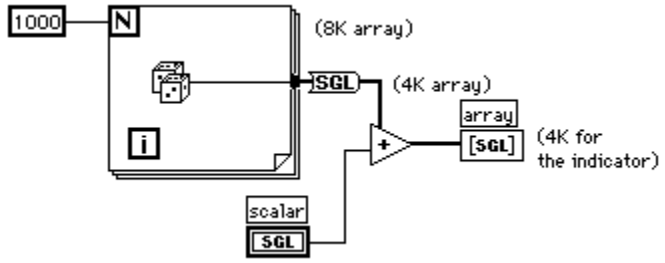
How to Generate Data of the Right Type

Consider the following example, which creates an array of 1,000 random values and adds it to a scalar. The coercion dot at the Add function occurs because the random data is double precision, while the

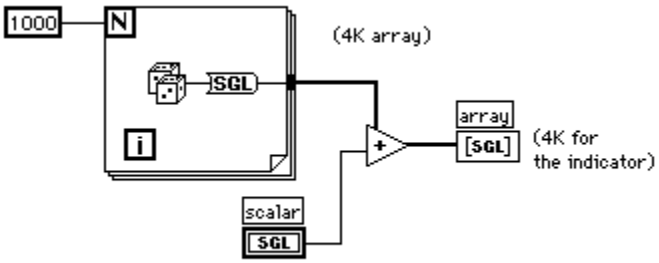
scalar is single precision. The scalar is promoted to a double precision before the addition. The resulting data is then passed to the indicator. This diagram uses up 16 KB of memory.



The following diagram incorrectly attempts to correct this problem by converting the array of double-precision random numbers to an array of single-precision random numbers. It uses the same amount of memory as the previous example.



The best solution, shown in the following illustration, is to convert the random number to single precision as it is created, before you create an array. Doing this avoids the conversion of a large data buffer from one data type to another.

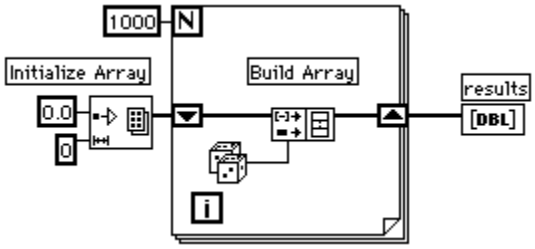


Avoid Constantly Resizing Data

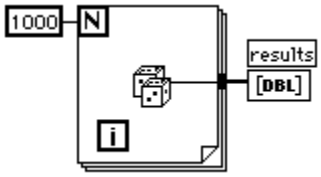
If the size of an output is different from the size of an input, the output does not reuse the input data buffer. This is the case for functions such as Build Array, String Concatenate, and Array Subset which increase or decrease the size of an array or string. When working with arrays and strings, try to avoid constantly using these functions, because your program uses more data memory, and executes more slowly because it is constantly copying data.

Example 1: Building Arrays

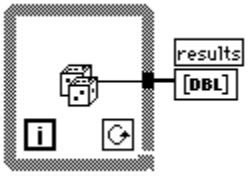
As an example, consider the following diagram which is used to create an array of data. This diagram creates an array in a loop by constantly calling Build Array to concatenate a new element. The input array is not reused by Build Array. Instead, LabVIEW continually resizes the output buffer to make room for the new array, and copies data from the old array to the new array. The resulting execution speed is very slow, especially if the loop is executed many times.



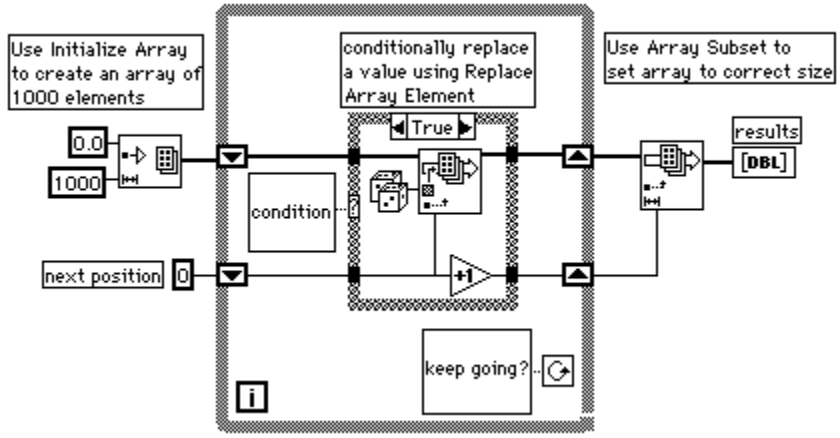
If you want to add a value to the array with every iteration of the loop, you will get the best performance by using autoindexing on the edge of a loop. With For Loops, LabVIEW can predetermine the size of the array (based on the value wired to N), and resize the buffer only once.



With While Loops, autoindexing cannot be quite as efficient, because the end size of the array is not known. However, While Loop autoindexing avoids resizing the output array every iteration by increasing the output array size in large increments. When the loop is finished, the output array is resized to the correct size. The performance of While Loop autoindexing is nearly identical to For Loop autoindexing.

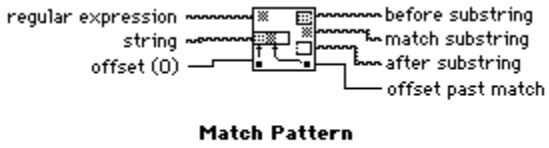


Autoindexing assumes that you are going to add a value to the resulting array with each iteration of the loop. If you need to conditionally add values to an array, but you can determine an upper limit on the array size, you might consider preallocating the array and then using Replace Array Element to fill up the array. When you are finished filling up the array values, you can resize the array to the correct size. The array is created only once, and Replace Array Element can reuse the input buffer for the output buffer. The performance for this is very similar to the performance of loops using autoindexing. If you use this technique, you need to be careful that the array in which you are replacing values is large enough to hold the resulting data, because Replace Array Element does not resize arrays for you.
An example of this process is shown in the following illustration.



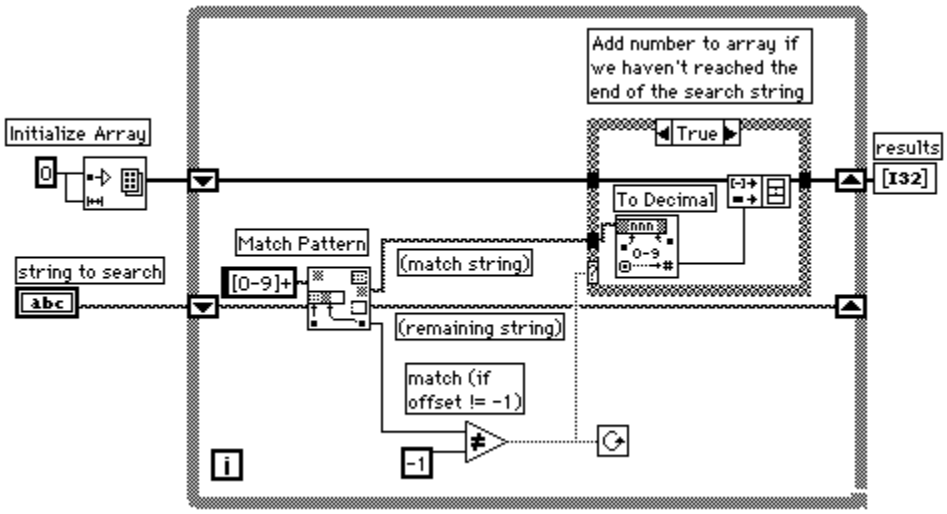
Example 2: Searching through Strings

You can use the Match Pattern function to search a string for a pattern. Depending on how you use it, you may slow down performance by unnecessarily creating string data buffers. The following illustration shows the Help window for Match Pattern.

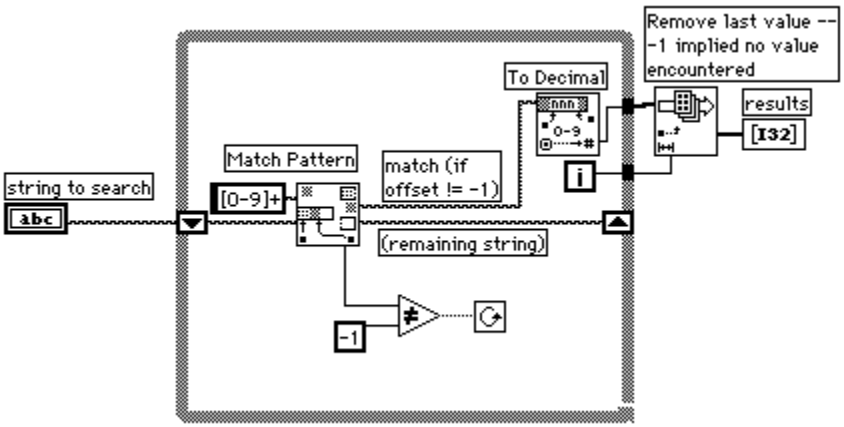


Assuming you want to match an integer in a string, you could use [0-9]+ as the regular expression input to this function. To create an array of all integers in a string, you need to use a loop and call Match Pattern repeatedly until the offset value returned is -1.

The following diagram is one method for scanning for all occurrences of integers in a string. It initially creates an empty array, and then each time through the loop, searches the remaining string for the numeric pattern. If the pattern is found (offset is not -1), this diagram uses Build Array to add the number to a resulting array of numbers. When there are no values left in the string, Match Pattern returns -1 and this diagram completes execution.

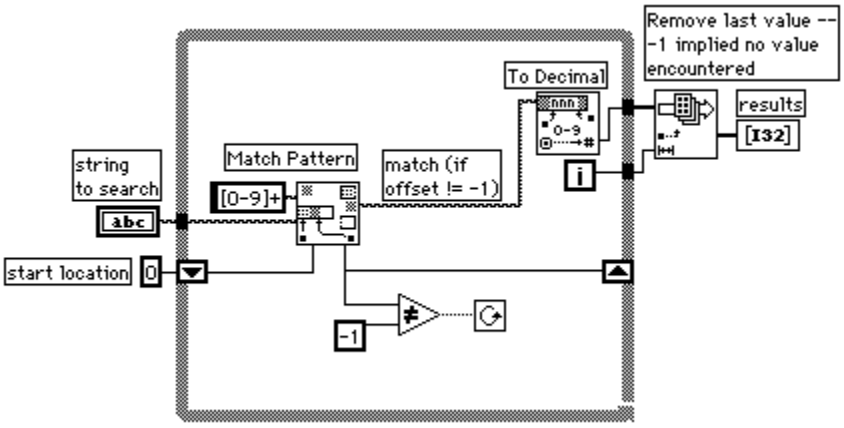


One problem with this diagram is that it uses Build Array in the loop to concatenate the new value to the previous value. Instead, you could use autoindexing to accumulate values on the edge of the loop. Notice that you end up getting an extra, unwanted value in the array from the last iteration of the loop where Match Pattern fails to find a match. A solution is to use array subset to remove the extra unwanted value. This is shown in the following illustration.



The other problem with this diagram is that you create an unnecessary copy of the remaining string every

time through the loop. Match Pattern has an input you can use to specify where to start searching. If you remember the offset from the previous iteration, you can use this number to specify where to start searching on the next iteration. This technique is shown in the following memory usage illustration.

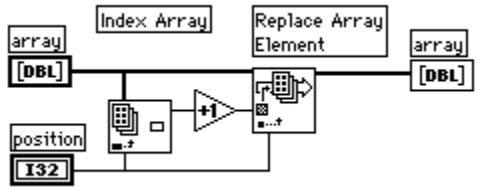


Efficient Data Structures

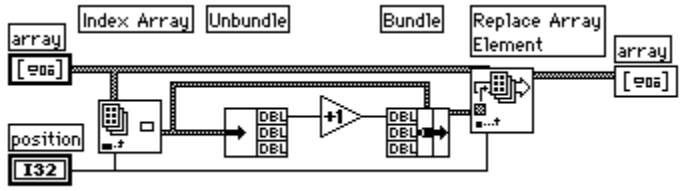
One of the points made in the previous section is that hierarchical data structures such as arrays of clusters containing large arrays or strings, or clusters containing large arrays or strings, cannot be manipulated efficiently. This section explains why this is so and gives strategies for choosing more efficient data types.

The issue with complicated data structures is that it is difficult to access and change elements within a data structure without causing copies of the elements you are accessing to be generated. If these elements are large, as in the case where the element itself is an array or string, these extra copies use more memory and the time that it takes to copy the memory.

You can generally manipulate scalar data types very efficiently. Likewise, you can efficiently manipulate small strings and arrays where the element is a scalar. In the case of an array of scalars, the following code shows what you would have to do to increment a value in an array.



This is quite efficient because no extra copies of the overall array need to be generated. Also, the element produced by the Index Array function is a scalar, which can be created and manipulated efficiently.
The same is true of an array of clusters, assuming the cluster contains only scalars. In the following diagram, manipulation of elements becomes a little more complicated, because you now have to use Unbundle and Bundle. However, because the cluster is probably small (scalars use very little memory), there is no significant overhead involved in accessing the cluster elements and replacing the elements back into the original cluster.



If you have an array of clusters where each cluster contains large sub-arrays or strings, indexing and

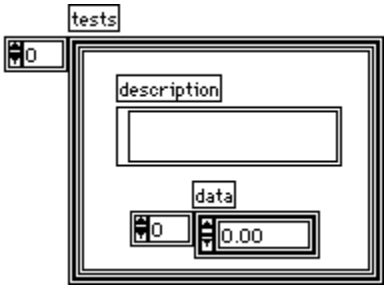
changing the values of elements in the cluster can be more expensive in terms of memory and speed. When you index an element in the overall array, a copy of that element is made. Thus, a copy of the cluster and its corresponding large subarray or string is made. Because strings and arrays are of variable size, the copy process can involve memory allocation calls to make a string or subarray of the appropriate size, in addition to the overhead to actually copy the data of a string or subarray. This may not be significant if you only plan to do it a few times. However, if your application centers around accessing this data structure frequently, the memory and execution overhead may add up quickly.

The solution is to look at alternative representations for your data. The following three case studies present three different applications, along with suggestions for the best data structures in each case.

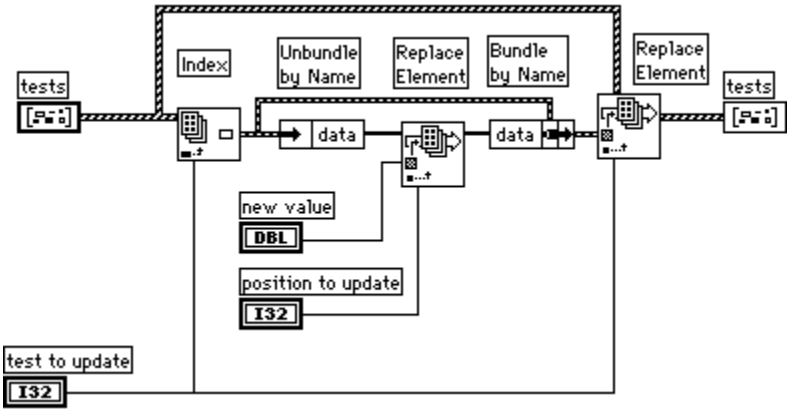
Case Study 1: Avoiding Complicated Data Types
Case Study 2: Global Table of Mixed Data Types
Case Study 3: A Static Global Table of Strings

Case Study 1: Avoiding Complicated Data Types

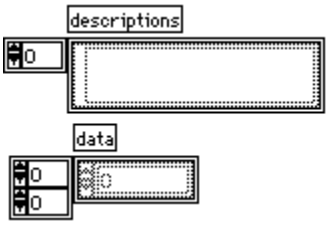
Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of the test results. One data type you might consider using to store this information is shown in the following illustration.



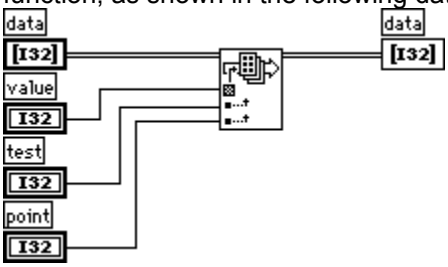
To change an element in the array, you have to index an element of the overall array. Now, for that cluster you have to unbundle the elements to get to the array. You then replace an element of the array, and store the resulting array in the cluster. Finally, you store the resulting cluster into the original array. An example of this is shown in the following illustration.



Each level of unbundling/indexing may result in a copy of that data being generated. Notice that a copy is not necessarily generated. The rules for when a copy is made are complex, and are discussed in LabVIEW Technical Note 020, Minimizing the Number of Data Buffers. Copying data is costly in terms of both time and memory. The solution is to try to make the data structures as flat as possible. For example, in this case study you could break the data structure into two arrays. The first array would be the array of strings. The second array would be a 2D array, where each row is the results of a given test. This result is shown in the following illustration.



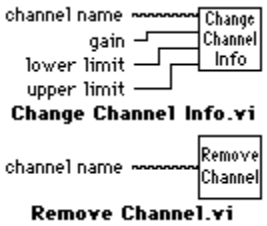
Given this data structure, you can replace an array element directly using the Replace Array Element function, as shown in the following data types:avoiding complicated data types in structures illustration.



Case Study 2: Global Table of Mixed Data Types

Consider another application in which you want to maintain a table of information. In this application, you decide you want the data to be globally accessible. This table might contain settings for an instrument, including gain, lower and upper voltage limits, and a name used to refer to the channel.

To make the data accessible throughout your application, you might consider creating a set of subVIs to access the data in the table, such as the following subVIs, the Change Channel Info VI and the Remove Channel Info VI.



The following section presents three different implementations for these VIs.

Obvious Implementation

Given this set of functions, there are several data structures you could consider for the underlying table. First, you might use a global variable containing an array of clusters, where each cluster contains the gain, lower limit, upper limit, and the channel name.

As described in the previous section, this data structure is difficult to manipulate efficiently, because you generally have to go through several levels of indexing and unbundling to access your data. Also, because the data structure is a conglomeration of several pieces of information, you cannot use the Search 1D Array function to search for a channel. You can use Search 1D Array to search for a specific cluster in an array of clusters, but you cannot use it to search for elements that match on a single cluster element.

Alternative Implementation 1

As with the previous example, you could choose to keep the data in two separate arrays. One could contain the channel names. The other array can contain the channel data. The index of a given channel name in the array of names is used to find the corresponding channel data in the other array.

Notice that because the array of strings is separate from the data, you can use the search 1D Array function to search for a channel.

In practice, if you are creating an array of 1,000 channels using the Change Channel Info VI, this implementation is roughly twice as fast as the previous version. This change is not very significant because there is other overhead that is affecting performance.

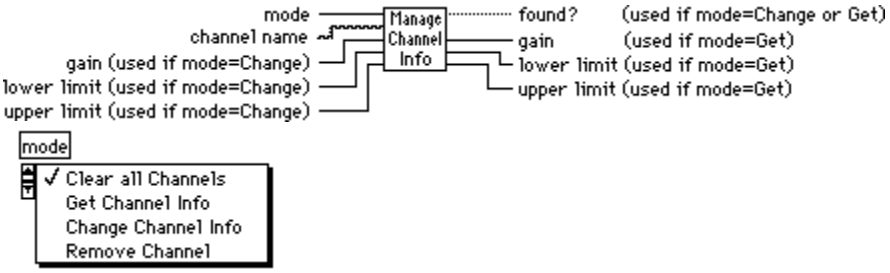
If you have read through the General Rules for Better Memory Usage topic, you may have seen a note indicating that you should avoid overusing local and global variables. When you read from a global variable, a copy of the global variable's data is generated. Thus, a complete copy of the array's data is being generated each time you access an element. The next method shows an even more efficient method that avoids this overhead.

Alternative Implementation 2

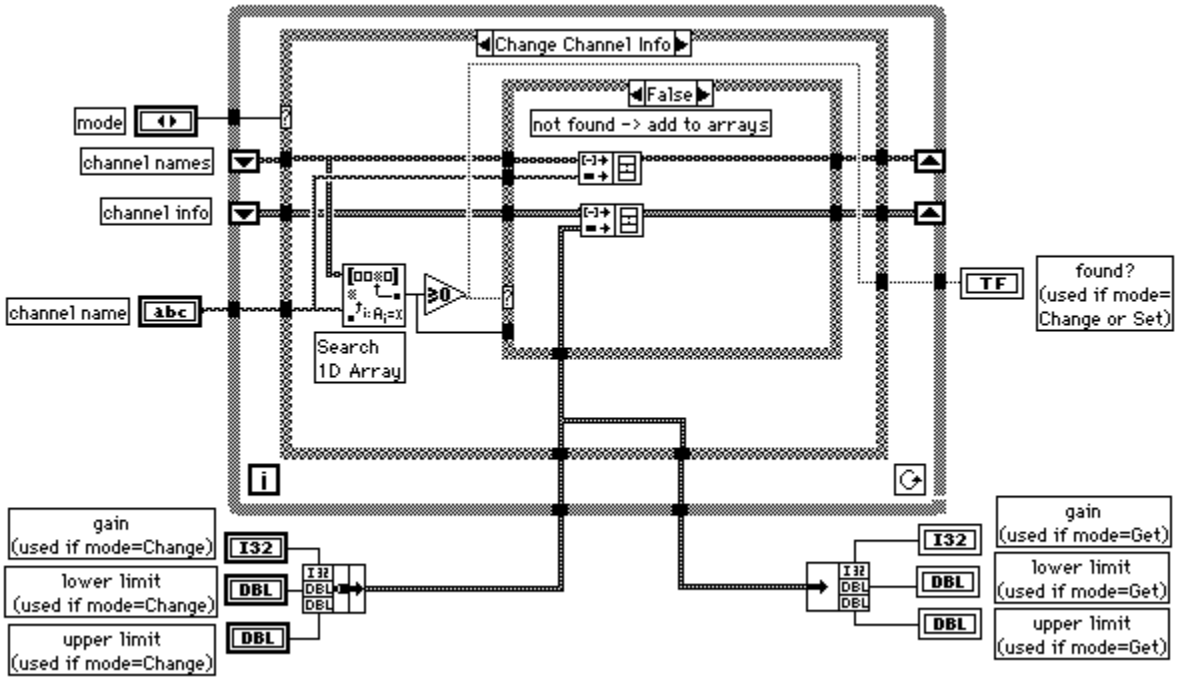
There is an alternative method for storing global data, and that is to use an uninitialized shift register. Essentially, if you do not wire an initial value, a shift register remembers its value from call to call. If you are not familiar with uninitialized shift registers, see Chapter 3*, Loops and Charts*, of the *LabVIEW Tutorial Manual* before continuing with this discussion.

LabVIEW handles access to shift registers efficiently. Reading the value of a shift register does not necessarily generate a copy of the data. In fact, you can index an array that is stored in a shift register and even change and update its value without generating extra copies of the overall array. The only problem with a shift register is that only the VI that contains the shift register can access the data of the shift register. On the other hand, this provides the advantage of modularity.

What you can do is make a single subVI with a mode input that specifies whether you want to read, change, or remove a channel, or whether you want to zero out the data for all channels, as shown in the following illustration.



The subVI would contain a While Loop with two shift registers--one for the channel data, and one for the channel names. Neither of these shift registers should be initialized. Then, inside the While Loop you would place a Case Structure connected to the mode input. Depending on the value of the mode, you could read and possibly change the data in the shift register.
Following is an outline of a subVI with an interface that could handle these three different modes. Only the Change Channel Info code is performance issues:efficient data structures:global table of mixed data typesshown.

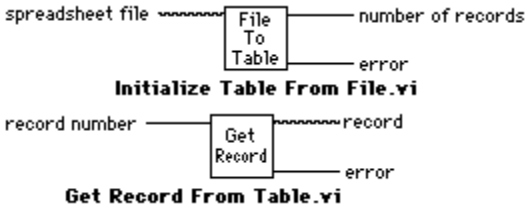


For 1,000 elements, this implementation is twice as fast as the previous implementation, and four times faster than the original implementation.

Case Study 3: A Static Global Table of Strings

The previous example looked at an application in which the table contained mixed data types, and the table might change frequently. In many applications, you have a table of information that is fairly static once created. The table might be read from a spreadsheet file. Once read into memory, you mainly use it to look up information.

In this case, your implementation might consist of the following two functions, Initialize Table From File and Get Record From Table.



One way to implement the table would be to use a two-dimensional array of strings. Notice that LabVIEW stores each string in an array of strings in a separate block of memory. If there are a large number of strings (for example, more than 5,000 strings), you may begin to put a load on the memory manager. This load can cause a noticeable loss in performance as the number of individual objects increases.
An alternative method for storing a large table is to read the table in as a single string. Then, build a separate array containing the offsets of each record in the string. This changes the organization so that instead of having potentially thousands of relatively small blocks of memory, you have instead one large block of memory (the string) and a separate smaller block of memory (the array of offsets).

This method may be more complicated to implement, but it can be much faster for large tables.

Portability Issues

This topic discusses issues related to transporting VIs between platforms.

Portable and Nonportable VIs
Ease of Porting Between Platforms

Portable and Nonportable VIs

VIs created with LabVIEW are portable among all the platforms LabVIEW runs on, as long as the versions of LabVIEW are the same (4.0, for example). VIs containing CINs or platform-specific features such as DDE are not portable (the VI ports, but it is broken).

LabVIEW uses the same file format on all platforms. You can transfer VIs from one system to another system, either by disk or over a network.

After the file is on the new system, you can open the VI from within LabVIEW on that platform. LabVIEW detects that the VI is from another platform and recompiles the VI to use the correct instructions for the current processor. If you transfer VIs on a disk formatted for a different platform, you may need to use a utility program such as Apple File Exchange on the Macintosh to read the disk.

You cannot port the following VIs:

- VIs distributed in the `vi.lib` directory. Each distribution of LabVIEW contains its own vi.lib, so do not move VIs in `vi.lib` across platforms.
- Compatibility VIs. You may be using compatibility VIs if you created VIs in earlier versions. Compatibility VIs emulate functionality found in older versions of LabVIEW. In many instances, these VIs use CINs that are only available on one platform and consequently cannot be ported to other platforms (because the VIs are broken).
- VIs containing CINs. You get an `object code not found` error if the CIN is from a different platform. If you write your CIN source code in a platform independent manner, you can recompile it on another platform and relink it to the ported VI.
- VIs containing the Call Library function. The VI will port, but is broken unless it can find a library of the same name.
- Platform-specific communication VIs such as AppleEvents on the Macintosh and DDE on Windows.
- The LabVIEW for Windows In Port and Out Port Utility VIs and the LabVIEW for Macintosh Peek and Poke Utility VIs.

Ease of Porting Between Platforms

Separator Character Differences
Resolution and Font Differences
Overlapping Labels
Picture Differences

There are several things you can do to ease porting between platforms. Portability issues include differences in filenames, separator characters, resolutions and fonts, possible overlapping of labels, and differences in picture formats.

One consideration is the filename. Filenames are limited to eight characters plus an optional three-character extension, typically .vi under DOS/Windows 3.x and under FAT volumes in Windows NT. LabVIEW for Macintosh filenames can have 31 characters. LabVIEW for Sun, LabVIEW for HP-UX, and LabVIEW for Windows NT/Windows 95 filenames can have 255 characters, including the .vi extension.

To avoid complications, you should either save VIs with short names, or save them into a VI library. A VI

library is a single file that can contain multiple VIs. A library name must conform to platform limits, but VIs in libraries can have names up to 255 characters, regardless of platform. Thus, VI libraries are the most convenient format for transferring VIs, because libraries eliminate most file system dependencies. See the Saving VIs in VI Libraries (.LLBs) topic for more information on creating VI libraries.

Separator Character Differences

If you are developing a VI for use on multiple platforms, you should not use any of the platform-specific path separator characters [(\ ), (/), and (:)] in your filenames. Avoid any special characters in your filenames because these can be interpreted differently by different file systems. For example, hidden files in UNIX begin with a period.

Resolution and Font Differences

Another portability issue concerns differences in screen resolution and fonts. Fonts can vary from platform to platform, so after porting a VI, you may have to choose new fonts to get an appealing display. When designing VIs, keep in mind that there are three fonts which best map between platforms--the Application font, the System font, and the Dialog font.

These predefined fonts and the real fonts they map to are as follows:

- The Application font is the LabVIEW default font. It is used in the **Controls** palette, the **Functions** palette, and new controls.
  - ***(Windows)*** The U.S. version of Windows usually uses Arial. The size depends on the settings of the video driver, because you can frequently set up higher resolution video drivers to use Large Fonts or Small Fonts. Under the Japanese version of Windows, LabVIEW uses the font that Windows uses for file names in the program manager.
  - ***(Macintosh)*** LabVIEW uses the same font used in the Finder for file names for the Application font. For example, on the U.S. Macintosh system, LabVIEW uses Geneva, while on the Japanese Macintosh system, LabVIEW uses Osaka.
  - ***(UNIX)*** LabVIEW uses Helvetica by default.

- The System font is the font LabVIEW uses for menus.
  - ***(Windows)*** Windows uses Helvetica, with the size dependent upon the video driver.
  - ***(Macintosh)*** Thus, LabVIEW normally uses Chicago under the U.S. system software, Osaka under the Japanese system software, and so on.
  - ***(UNIX)*** LabVIEW normally uses Helvetica for this font.
- The Dialog font is the font LabVIEW uses for text in dialog boxes.
  - ***(Windows)*** Under the Japanese version of Windows, this font is the same as the System font. Under the U.S. version of Windows, this font is a bold version of the Application font.
  - ***(Macintosh)*** This font is the same as the System font.
  - ***(UNIX)*** On HP-UX, this font is the same as the System font. LabVIEW normally uses Helvetica for this font.

When you take a VI that contains one of these fonts to another platform, LabVIEW ensures that the font maps to something reasonable on that platform.

If you do not use the predefined fonts, but instead select a specific font such as Geneva or New York, the font can change size on the new platform because of the differences in the fonts available and differences in the resolution of the display. If you select Geneva or New York on the Macintosh, LabVIEW cannot match it on the Sun or HP-UX, and LabVIEW uses the font named *fixed*. If you take a VI with

unrecognized fonts to Windows, you may not get a good mapping to a new font.

If you use a predefined font on a section of text, National Instruments recommends that you do not change the size of that text. If you change the font to something other than the default size and then take the VI to a different platform, LabVIEW tries to match the font with the new size, which may be inappropriate given the resolution of the screen. For example, an application font with size 10 (one pixel bigger than the default) looks good on the Macintosh (Geneva 10) but looks very tiny on Windows with a high-resolution video driver, where it is three pixels smaller than the default.

Overlapping Labels

When you move a VI to a new platform, controls and labels may change size, depending on whether the fonts are smaller or larger. LabVIEW tries to keep labels from overlapping their owners by moving them away from the owning control. Also, every label and constant has a default attribute called **Size to Text**. When you first create a label or constant, this attribute is set, so the bounds of the object resize as necessary to display all of the enclosed text.

If you ever manually resize the object, LabVIEW turns off this attribute (the item in the pop-up menu is no longer checked). With **Size to Text** turned off, the bounds of the object stay constant, and LabVIEW clips (or crops) the enclosed text as necessary. If you do not want LabVIEW to clip text when you move between systems or platforms, you should leave this attribute on for labels and constants.

Most Sun and HP monitors are much larger and have a higher resolution than PC and Macintosh monitors. Sun and HP-UX users should not make their front panels very large if they want them to port well.

For best results, avoid overlapping controls and leave extra space. If a label even partially overlaps another object and the font grows, it may end up overlapping the control.

Picture Differences

The most basic type of picture contains a bitmap, a series of values specifying the color of each pixel in the picture. More complex pictures may contain any number of commands that are executed every time the picture is drawn. Pictures containing drawing commands are created by drawing programs or in the draw layer of a graphics application. Bitmap-based pictures are created by paint programs or in the paint layer of a graphics application.

Bitmaps are common storage formats for pictures on all platforms. If you use pictures that contain bitmaps on your front panels, the pictures usually look the same when you load your VIs on another platform. However, pictures containing drawing commands might include some commands that are not supported on other platforms, for example clipping and pattern filling. These pictures might look strange on other platforms. You should always see how your VIs look on another platform if you expect them to be used there.

You can still use a drawing program or the draw layer of a graphics application to create your pictures. But to make them more portable, paste the final picture into a paint program or into the paint layer of a graphics application before copying the picture into LabVIEW.

(Windows 95, Windows NT, and Macintosh) Many graphics applications on the Macintosh and certain applications on Windows 95 and Window NT allow you to cut or copy a picture with a non-rectangular shape. For example, you can use a Lasso tool to select the outline of a circle, triangle, or a more complicated shape like a musical note

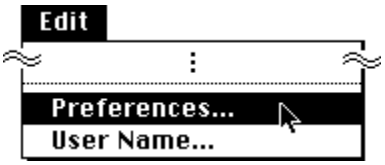
 . Other platforms might have to draw these irregular shapes on a rectangular white background. Under Windows 95 or Windows NT, look for applications that support enhanced metafiles if you want

pictures with irregular shapes and pictures that scale well.

Setting LabVIEW Preferences

To customize LabVIEW and configure some default parameters, you use the various Preferences dialog boxes.

You access the Preferences dialog boxes by selecting **Edit»Preferences…**, as shown in the following illustration.



Use the pull-down menu at the top of the dialog box that appears to select among the different categories of preferences, as shown in the following illustration.

To learn about any of these Preferences, select an item from the following list:

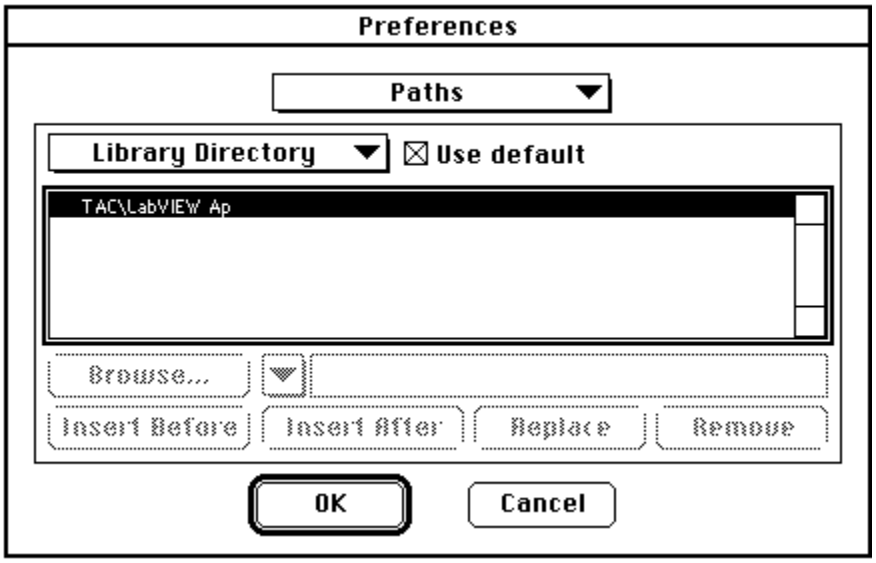
Path Preferences
Performance and Disk Preferences
Front Panel Preferences
Block Diagram Preferences
Debugging Preferences
Color Preferences
Font Preferences
Printing Preferences
History Preferences
Time and Date Preferences
Miscellaneous Preferences

To learn how the preferences are stored, select the following topic:

How Preferences are Stored

Path Preferences

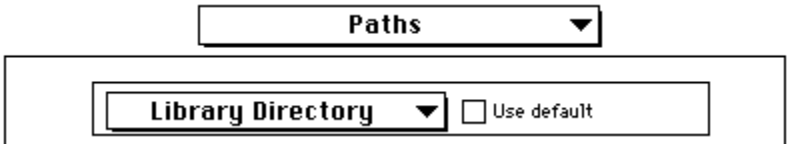
You can specify the directories that LabVIEW searches when looking for VIs as well as the paths LabVIEW uses for temporary files and the library directory. If not already selected, select **Paths** from the pull-down menu in the Preferences dialog box to bring up the dialog box shown in the following illustration.



Note: The format of pathnames--the use of colons, slashes, or backslashes--differs slightly on different platforms.

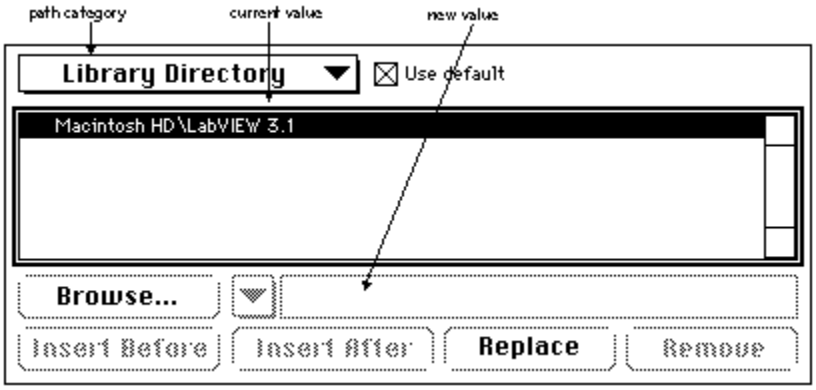
In the Path Preferences dialog box is another pull-down menu, shown in the following illustration, from which you select the path category you want to view or edit.

In the dialog box that appears, many items are grayed-out and inactive because LabVIEW is set to use the default path. If you want to change one of these preferences, deselect the **Use Default** checkbox, as shown in the following illustration.



Library, Temporary, and Default Directories

The **Library Directory**, **Temporary Directory**, and **Default Directory** are single directories (folders). When you edit one of these paths, you are given options to type in a new path and replace the existing path, or browse the file dialog box to select a path. This is shown in the following illustration.



The **Library Directory** specifies the absolute pathname to the directory containing `vi.lib` and any library directories you supply. The default is the directory containing LabVIEW.
The **Temporary Directory** specifies the absolute pathname to the directory for temporary files, which varies according to platform, as follows:

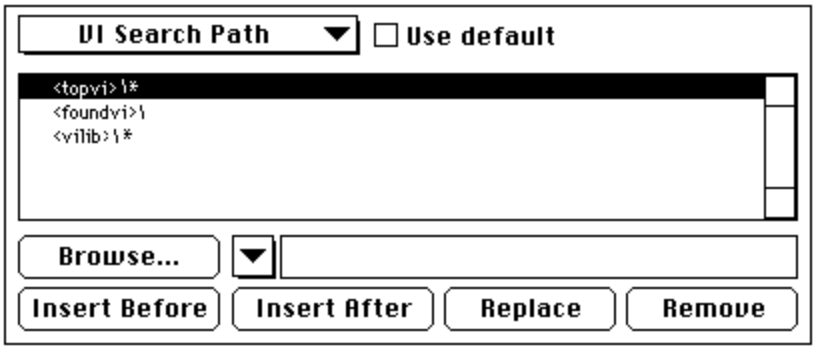
- ***(Windows)*** The default is the directory containing LabVIEW.
- ***(Macintosh***) In System 6, the default is the LabVIEW folder. Under System 7, the default is an invisible folder called `Temporary Items` at the top of your hard drive, which is where Apple recommends that temporary files be stored. If LabVIEW crashes, the temporary files are moved into the trash when you restart.
- ***(UNIX)*** The default is the `/tmp` directory.

The **Default Directory** specifies the absolute pathname to the default directory, which is the initial directory the file dialog box displays. The Default Directory function in the **Functions»File I/O»File Constants** palette also returns this value. The default is the current working directory. Changes to the Default Directory take place immediately.

Note: All changes to the Library Directory, Temporary Directory, and Default Directory options take effect when you relaunch LabVIEW.

VI Search Path

The VI Search Path is used only when LabVIEW searches for a subVI, control, or external subroutine that is not in the expected location. It lets you list the path of directories for LabVIEW to search. When you edit the search path, you are given more options than for the other paths--you can add new items in specific locations, remove paths, and select from a list of *special* paths.



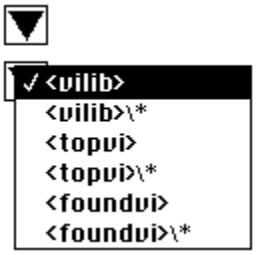
The list indicates paths that LabVIEW searches, in the order that LabVIEW searches them. To add a new directory, you first need to decide when LabVIEW should search that directory relative to the other directories. Select an adjacent directory from the list. Then use either the browse button, which displays a file dialog box, or the special pull-down menu, which gives a list of special paths, to select the directory that LabVIEW should search. You can also edit or enter this path in the string control next to these options. Finally, use **Insert Before**, **Insert After**, or **Replace** to add that option to the list.

Notice that when you select a path, LabVIEW normally searches that directory, but not its subdirectories. You can make the search hierarchical by appending a * as a new path item.

Some examples on different platforms:

- ***(Windows)*** To search the directory `C:\VIs\` recursively, you would enter `C:\VIs\*`
- ***(Macintosh)*** To search recursively the VIs folder on the HD disk, you would enter `HD:VIs:*`
- ***(UNIX)*** To search recursively `/usr/home/gregg/vis`, you would enter `/usr/home/gregg/vis/*`.

You can use the special pull-down menu shown in the following illustration, which is located to the right of the **Browse...** button, to select from several special directories.



These directories are as follows:

- `<vilib>` as a path entry is a symbolic path that refers to the `vi.lib` directory inside of the Library directory.
- `<topvi>` is a symbolic path that refers to the directory containing the top level VI.
- `<foundvi>` refers to a list of directories that LabVIEW builds each time a VI is loaded. During a search, any directory in which LabVIEW finds a subVI, or that you manually select, is added to this list. Use this symbolic path so that if you move or rename a directory of VIs, and then open a calling VI, you have to manually find that directory only once for that load.

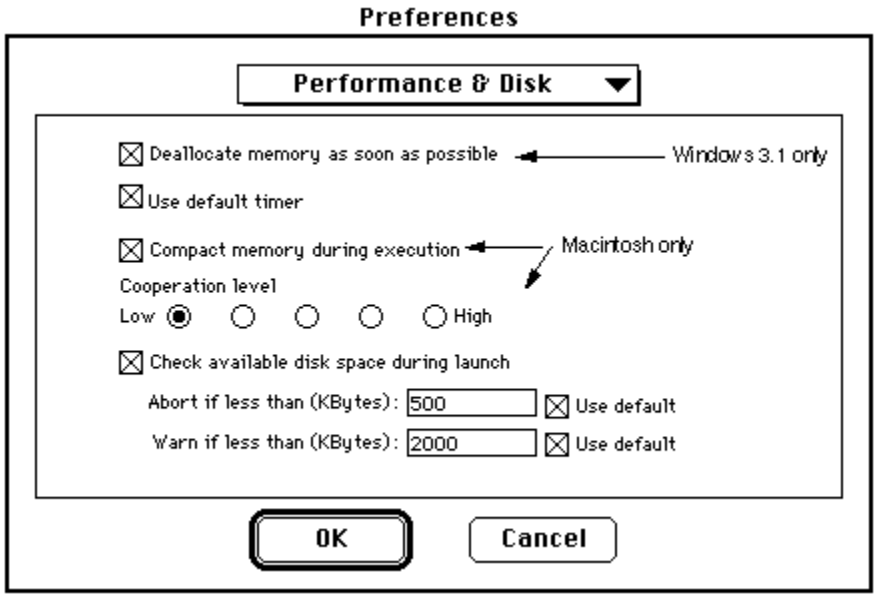
You can also remove a path using the **Remove** button, which places the removed path into the string box, in case you decide to re-insert it elsewhere in the list.

Note: Changes to the VI Search Path option take effect immediately.

Performance and Disk Preferences

While it executes, LabVIEW maintains temporary files on disk. For example, when you save a VI, LabVIEW first saves it into a temporary file. If the save is successful, the original is replaced with the new file. LabVIEW needs a fair amount of disk space for these operations. To avoid problems, LabVIEW normally checks for available disk space in the temporary directory at launch time. If you have less than 500 KB of disk space free, LabVIEW alerts you and quits. If you have less than 2 MB available, LabVIEW warns you, but continues. You can turn these warnings off or change the levels at which they alert you using the Performance and Disk dialog box, accessed from the Preferences pull-down menu.

The Performance and Disk Preferences dialog box is shown in the following illustration. Notice that one option is only available on Windows 3.1, and some other options are only available on the Macintosh.



Except for the **Deallocate memory as soon as possible** option, changes to the Performance & Disk options take place when you restart your computer.
The options in this dialog box are as follows:
Deallocate memory as soon as possible--Forces LabVIEW to deallocate the memory of a VI after it completes execution. Doing this may improve memory usage in some applications because subVIs deallocate their memory immediately after executing. However, it probably slows performance because LabVIEW has to allocate and deallocate memory more frequently.
Note: Changes to the Deallocate memory as soon as possible option take effect immediately.

(Windows 3.1) **Use default timer**--The timing functions in LabVIEW use the built-in Windows timer. This timer has, by default, a resolution of 55 ms (it increments 18 times a second). This means that the Tick Count, Wait, and Wait Until Next ms Multiple VIs in LabVIEW only have 55 ms resolution. Notice, however, that you can specify more accurate timing rates to the DAQ VIs, because they use the built-in timers of the DAQ boards to control the acquisition rate.

You can increase the resolution of the Tick Count, Wait, and Wait Until Next ms Multiple VIs to 1 ms by increasing the resolution of the built-in timer of Windows to 1 ms. If you increase the timer resolution, your software timing loops are more accurate.

Under certain circumstances, you may not want to increase the timer resolution from 55 ms. Changing the resolution increases the number of timer tick interrupts from 18 interrupts per second to 1000 interrupts per second. If you are performing other interrupt intensive operations, you may exceed the capacity of your PC to handle the interrupt load, which may lock up or crash your PC.

DAQ operations that use programmed I/O instead of DMA to transfer data between the DAQ board and PC memory are interrupt intensive. You should not change the timer tick resolution to 1 ms if you plan to perform any of the following operations at high transfer rates:

- Buffered analog input using the PC-LPM-16
- Buffered analog output using the AT-MIO-16, MC-MIO-16, or Lab series boards
- Buffered digital I/O using the DIO-24, DIO-96, or Lab series boards
- Buffered analog input or output with DMA disabled in WDAQCONF using any DAQ board

Transfer rates of approximately 30 ksamples/sec or higher on a 386 25 MHz machine causes interrupt load problems with the 1 ms timer resolution. Transfer rates of approximately 75 to 80 ksamples/sec or higher on a 486 33 MHz machine, or 100 ksamples/sec or higher on a 486 50 MHz machine, may cause interrupt load problems.

The timing functions in LabVIEW for Windows 95 and Windows NT have a timer resolution of 1 ms, the maximum resolution possible under Windows 95 or Windows NT. Slower machines may have lower resolution.

(Macintosh) **Compact memory during execution**--Directs LabVIEW to compact memory approximately once every 30 seconds. Compaction reduces fragmentation by moving allocated memory to a single location. A disadvantage to this option is that compaction takes time and may cause a short lull in performance while LabVIEW is compacting memory.

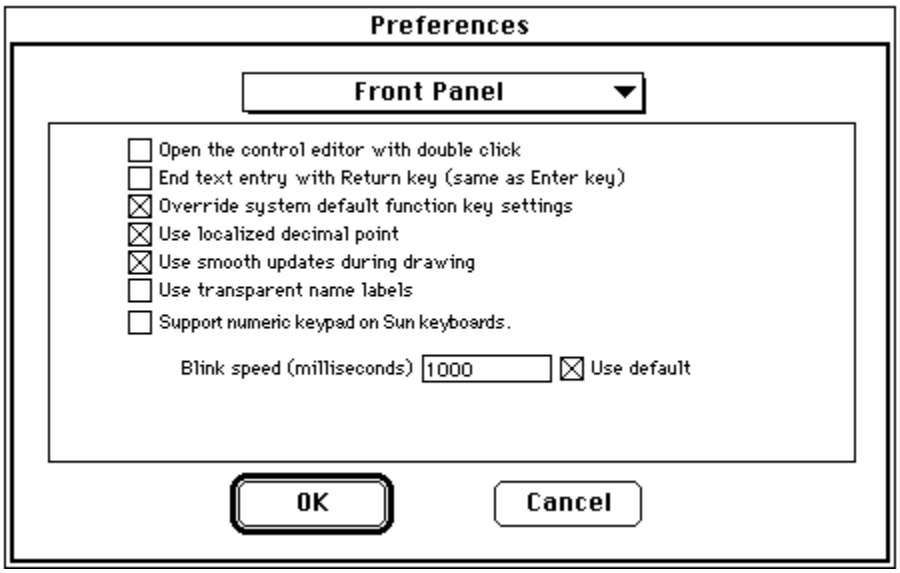
(Macintosh) **Cooperation level**--Lets you control how cooperative LabVIEW is with other background applications. If you set it to **Low**, LabVIEW does not share time very frequently with other applications. Setting it low improves the performance of LabVIEW, but causes the performance of any other applications that run in parallel to decrease. If you set it to **High**, LabVIEW performance suffers, but other applications have more time to execute.

Check available disk space during launch--Checks amount of disk space available in the temporary directory as LabVIEW launches. You can check whether you want LabVIEW to use the default in two cases: abort if less than 500 KB, or warn if less than 2,000 KB.

Note: Changes to the Check available disk space during launch option take effect when you relaunch LabVIEW. Changes to the Use default timer, Compact memory during execution, and Cooperation level options take effect immediately.

Front Panel Preferences

The Front Panel Preferences dialog box is shown in the following illustration:



The options in this dialog box are as follows:

Open the control editor with double click--Allows easy access to the Control Editor window, where you can customize the appearance of a front panel control or indicator.

End text entry with Return key (same as Enter key)--Makes the `<Enter>` (Windows); `<return>` (Macintosh); `<Return>` (Sun); or `<Enter>` (HP-UX) on the alphanumeric keyboard function like the `<Enter>` key on the numeric keypad, ending text entry. If you select this option, you can embed newlines by pressing `<Ctrl-Enter>` (Windows); `<option-return>` (Macintosh); `<meta-Return>` (Sun); or `<Alt-Return>` (HP-UX).

(Windows and Macintosh) **Override system default function key settings**--By default, your operating system may use some function keys for system purposes. For example, on the PC, pressing F10 is the same as pressing the `<Break>` key. On the Macintosh, F1 to F4 are usually treated as **Cut**, **Copy**, **Paste**, and **Clear**. If you turn this Override option on, the function keys are not used for system purposes but are passed instead to LabVIEW as standard function keys.

Use localized decimal point--Uses the systemÕs decimal separator instead of the period. For example, in many countries the comma is used as a decimal point. You would turn this option on if you want LabVIEW to pay attention to the way you have configured your system. You would turn it off if you want LabVIEW to use periods in all cases for the decimal point.

Use smooth updates during drawing--When LabVIEW updates a control with smooth updates off, it erases the contents of the control and draws the new value. This can result in a noticeable flicker as the old value is erased and replaced. Using smooth updates, LabVIEW draws data to an offscreen buffer and then copies that image to screen instead of erasing a section of the screen. This avoids the flicker caused by erasing and drawing. However, it can slow performance, and it requires more application memory because an offscreen drawing buffer has to be maintained.

Use transparent name labels--Directs LabVIEW to use labels that you can see through.

(Sun) **Support numeric keypad on Sun keyboards**--Turns on support for the Sun keyboard, including keypads, arrow keys, and the Help key. Do not turn this option on if you are using a non-Sun keyboard (for example, if you are using an X terminal or a PC running X software).

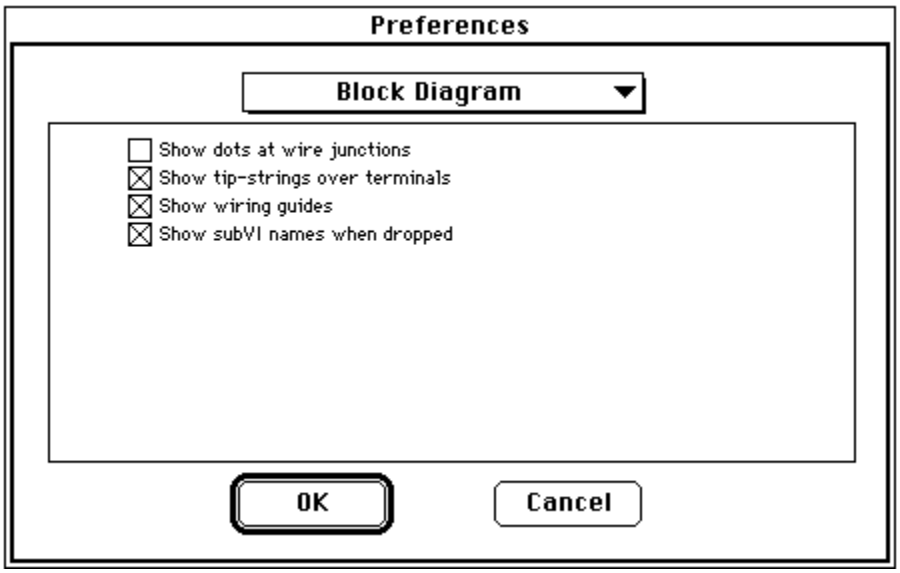
Blink speed--Sets the blinking speed for front panel objects, to the default 1,000 milliseconds or a speed that you type in. Blinking is a basic attribute turned on with Attribute Nodes. See Attribute Nodes for more

information.

Note: Changes to options in the Front Panel Preferences dialog box take effect immediately.

Block Diagram Preferences

The Block Diagram Preferences dialog box is shown in the following illustration:



The options in this dialog box are as follows:

Show dots at wire junctions--Directs LabVIEW to place dots wherever two wires cross, making it easier to differentiate between such junctions and wires that branch.

Show tip-strips over terminals--Directs LabVIEW to place parameter names over terminals when you idle your cursor over them in functions and subVIs.

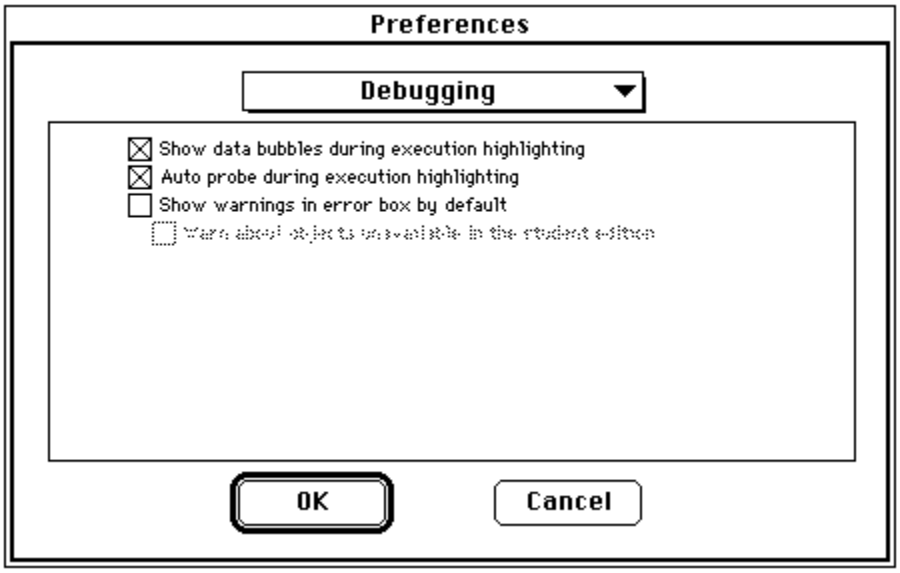
Show wiring guides--Shows wire stubs that indicate data type and data direction for each parameter when the cursor is idled over a block diagram node.

Show subVI names when dropped--Directs LabVIEW to label a subVI with the name of the original VI when you drop it on the block diagram, and to show the label.

Note: Changes to options in the Block Diagram Preferences dialog box take effect immediately.

Debugging Preferences

The Debugging Preferences dialog box is shown in the following illustration:



The options in this dialog box are as follows:

Show data bubbles during execution highlighting--Animates execution flow by drawing bubbles along the wires. You may want to turn this feature off if it significantly slows down performance on your computer.

Auto probe during execution highlighting--Probes scalar values automatically, drawing their values on the diagram. You can turn off this feature if you find that it clutters the display.

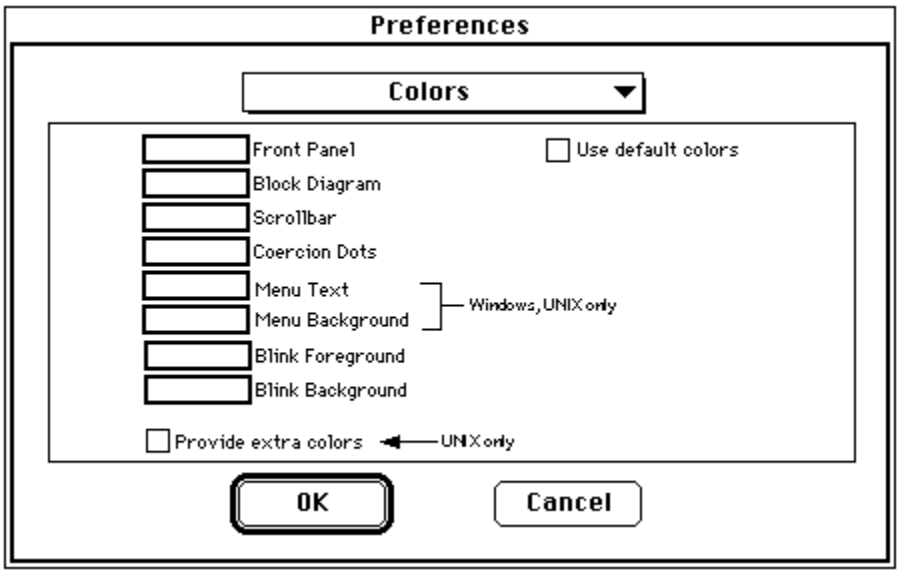
Show warnings in error box by default--Displays warnings in addition to errors in the Error List dialog box. A warning does not mean that the VI is incorrect; it just points out a potential problem in your block diagram.

Warn about objects unavailable in the student edition--If you have selected the previous option about warnings, this option includes warnings about objects unavailable in the LabVIEW Student Edition.

Note: Changes to options in the Debugging Preferences dialog box take effect immediately.

Color Preferences

The Color Preferences dialog box is shown in the following illustration:



This dialog box lets you change the colors used by LabVIEW for various items. If you do not select the **Use default colors** checkbox, you can click on any rectangle to change the color.
The options are the following:
Front Panel--Lets you select a color for the front panel of new VIs. Changing the color does not affect old VIs.
Block Diagram--Lets you select a color for the block diagram of new VIs. Changing the color does not affect old VIs.
Scrollbar--Lets you select a color for scrollbars. Affects only the VIs that are currently open.
Coercion Dots--Lets you select a color for the dots indicating coercion of numerical data. Affects only the VIs that are currently open.
(Windows, UNIX) **Menu Text**--Lets you select a color for text used in menus.
(Windows, UNIX) **Menu Background**--Lets you select a color for the background used in menus.

Blink Foreground--Lets you select the foreground color for a blinking object. Affects only a blinking object in its blink state. Blinking is a basic attribute turned on with Attribute Nodes. See Attribute Nodes for more information.

Blink Background--Lets you select the background color for a blinking object. Affects only a blinking object in its blink state.

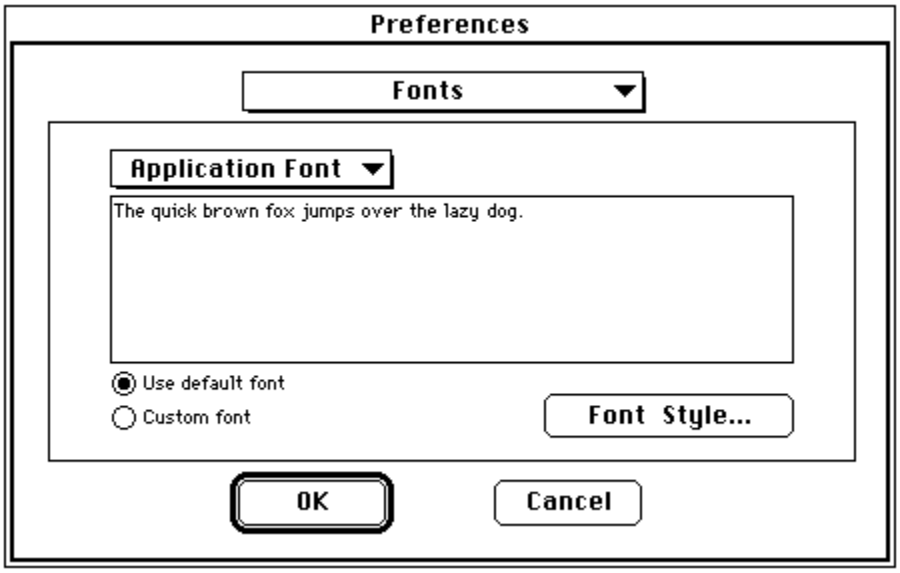
Use default colors--Uses LabVIEW's choice of colors for all items listed in this dialog box. You must turn this off if you wish to change any of these colors.

(UNIX) **Provide extra colors**--Toggles between a 216-color palette and a 125-color palette. A larger palette may cause a noticeable flash when you switch between windows.

Note: Changes to options in the Color Preferences dialog box take effect immediately.

Font Preferences

The Font Preferences dialog box lets you change three categories of predefined fonts: the Application font, System font, and Dialog font. You select the category of font you wish to change from the pop-up menu above the text box. The dialog box with Application Font selected is shown in the following illustration:



LabVIEW uses the three categories of fonts for specific portions of its interface. The fonts are predefined according to the platform, as follows:

Use default font--Uses the default font, as defined by LabVIEW in keeping with the different platforms, as described previously.

Custom font--This checkbox is automatically checked if you change any font characteristics through the Font Style... option.

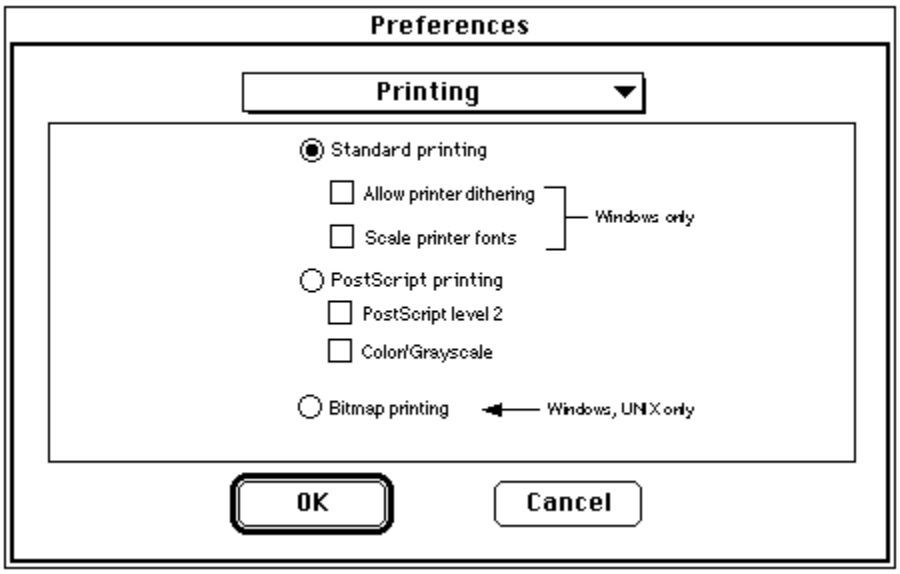
Font Style...--Brings up a dialog box that lets you change font characteristics.

Note: Changes to options in the Font Preferences dialog box take effect immediately.

For specific information on how the fonts are predefined or for information about the portability of fonts, see Portability Issues.

Printing Preferences

The Printing Preferences dialog box is shown in the following illustration. Note that some options are available on certain platforms only.



(Windows, Macintosh) **Standard printing**--Directs LabVIEW to format the VI print data (front panel, diagram, icon, etc.) and print it using standard drawing commands. Use this option if you want to print to a file (assuming your printer driver supports printing to a file), if your printer does not have PostScript support, or if you want your printer driver to do the PostScript translation instead of LabVIEW.

(Windows only) **Allow printer dithering**--Directs LabVIEW to use printer dithering when creating the drawing commands. Applicable for black and white printing, printer dithering is a method of adding gray scales to an image instead of only black and white areas. This option is not dependent on printer type.

(Windows only) **Scale printer fonts**--Enables a more complicated algorithm to select a different font if the first one doesn't scale correctly. If all of the text on your printed VI does not appear, or if it is too large, you can try this option to see if it improves the appearance of the printout. This option is not dependent on printer type.

PostScript printing--Directs LabVIEW to translate the VI print data in PostScript (.PS) format and send it to the printer as PostScript text. If you have a PostScript printer and a PostScript printer driver, the printout will be a graphics image. Do not select this option if your printer or printer driver does not support PostScript printing or you will get a printout containing nonsense.

PostScript level 2--Tells LabVIEW that the connected printer supports PostScript level 2. If this box is checked, LabVIEW sends PostScript level 2 code to the printer. If the box is not checked, LabVIEW sends PostScript level 1 code. PostScript level 1 is a subset of PostScript level 2. Select this option only if your printer supports PostScript level 2.

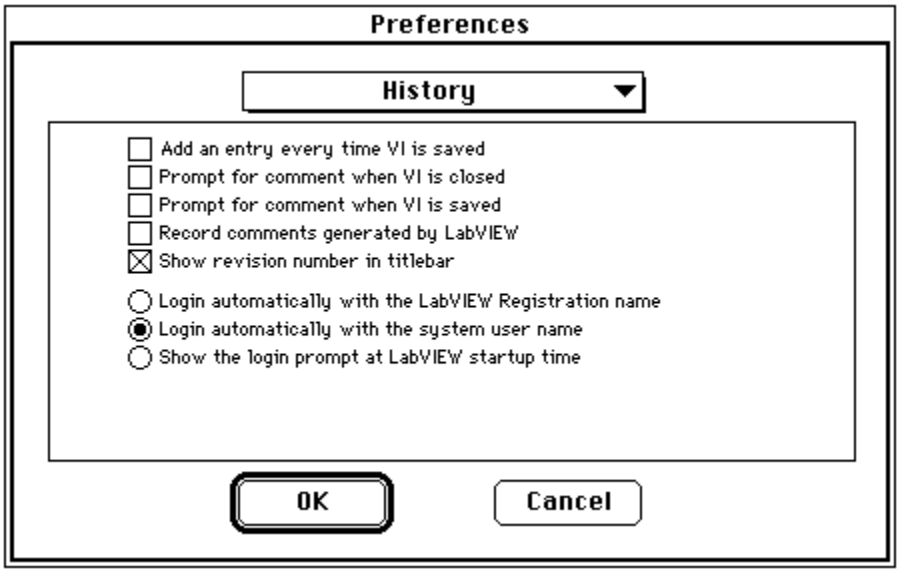
(Windows, UNIX) **Bitmap printing**--Directs LabVIEW to create a bitmap, draw all data for that page into the bitmap, and then send the bitmap to the printer. This method takes longer to print than the other two, but may yield a more accurate representation of the text and fonts than standard printing, even though the printout will not be as high resolution as with PostScript printing. The bitmap image is in color for color printers and black and white otherwise.

Note: Changes to options in the Printing Preferences dialog box take effect immediately.

History Preferences

As explained in Application Management,each VI has a History window that displays the development history of the VI. With the History Preferences dialog box, you can choose the default settings for the History window of new VIs.

The History Preferences dialog box is shown in the following illustration:



The options in the History dialog box are divided into two groups. The first group contains five options for specifying when and how entries to the History window are added for new VIs.

Note: Options concerning History window entries can be specified for individual VIs through the VI Setup…»Documentation dialog box. See Setting Documentation Optionsfor information about accessing the options.

The options in this dialog box are as follows:

Add an entry every time VI is saved--Directs LabVIEW to add an entry to the VI history every time you save the VI. If you have not entered a comment in the Comment box of the History window, only a header will be added to the history. (The header will contain the revision number if that option further down in this dialog box is checked, the date and time, and the name of the VI.)

Prompt for comment when VI is saved--Directs LabVIEW to bring up the History window whenever you save so that you can enter a comment. This is useful if you prefer to comment on your changes when you finish making them instead of as you are editing. If you do not set this option, you will not have a chance to change the history of the VI after you select Save until the save is finished.

Prompt for comment when VI is saved--Similar to the previous option, except that LabVIEW will only prompt you when you close a VI, rather than every time you save. You will be prompted if the VI changed any time since you loaded it, even if you have already saved the changes.

Note: You will not be prompted to enter a comment when you save or close a VI if the only changes you made were changes to the history.

Record comments generated by LabVIEW--Causes LabVIEW to insert comments into the History window when certain events occur. The events that cause automatic comments are conversion to a new version of LabVIEW, subVI changes, and changes to the name or path of the VI.

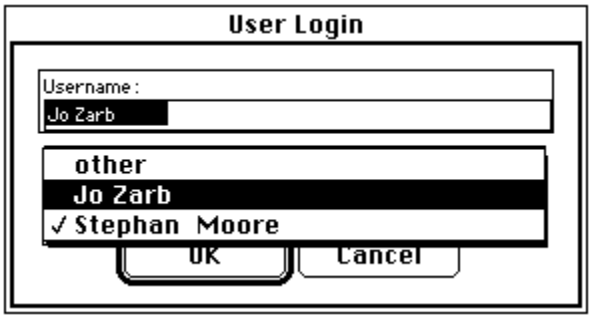
Show revision number in titlebar--Adds the revision number to the header of the History window. The

revision number starts at zero and is incremented every time you save the VI. It will not be incremented however, if the only changes you made were changes to the VI history.

The second group of options in the History Preferences dialog box concerns how you log in to LabVIEW, which in turn determines the name LabVIEW inserts in the headers to comments entered in the VI History window. The log-in options are as follows:

(Macintosh and Windows only) **Login automatically with the LabVIEW registration name**--Uses the name of the registered LabVIEW user.

Show the login prompt at LabVIEW startup time--Prompts you for a user name when LabVIEW is started. The user name can be changed any time by selecting **Edit»User Name**; the prompt displays a menu of all user names so-far entered into LabVIEW with this option.



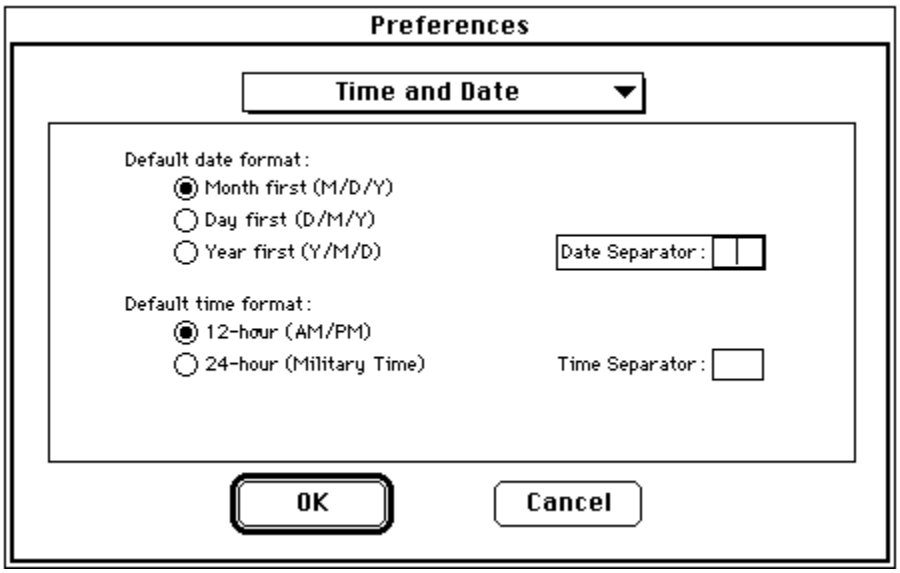
You can switch to another user name (the system user name or the LabVIEW registration name) in Windows or on the Macintosh, or the system user name in UNIX, without having to type it in by using the pull-down menu shown in the preceding illustration. You can also type in a name, which will appear in the list thereafter.

(Sun, HP-UX, Windows NT, and the Macintosh with file sharing only) **Login automatically with the system user name**--Assuming that a user name has been defined, causes LabVIEW to use the system login name of the current user.

Note: Changes to options in the History Preferences dialog box take effect immediately.

Time and Date Preferences

The Time and Date Preferences dialog box is shown in the following illustration:



The options in this dialog box control the default displays of time and date in the digital displays of new controls and indicators. You can override the defaults in this dialog box for individual digital displays, as explained in Format and Precision Changes of Digital Displays.

The options in this dialog box are as follows:

Default date format--Determines whether the month, day, or year comes first in digital displays.

Default time format--Determines whether time in digital displays is based on a 12-hour or 24-hour clock.

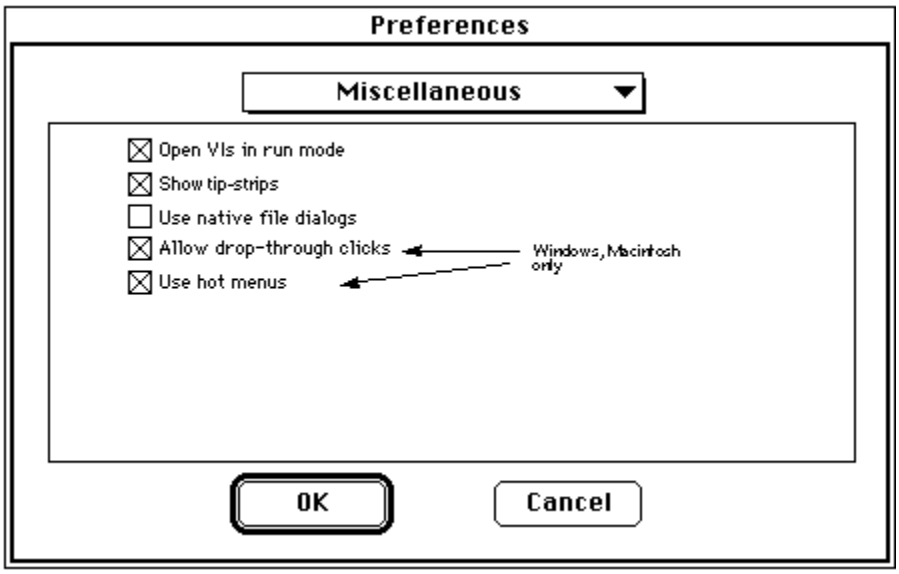
Date Separator--Determines the character used to separate the month, day, and year (in the order selected in the Default date format option) of digital displays.

Time Separator--Determines the character used to separate the hours and minutes past the hour in digital displays.

Note: Changes to options in the Time and Date Preferences dialog box take effect immediately.

Miscellaneous Preferences

The Miscellaneous Preferences dialog box is shown in the following illustration:



The options in this dialog box are as follows:

Open VIs in run mode--Opens VIs in run mode rather than edit mode.

Show tip-strips--Toggles the display of tip-strips.

(Windows, Macintosh) **Use native file dialogs**--Uses the operating system's native file dialog boxes in LabVIEW so that they look similar to those of other applications on your machine (with a few exceptions discussed at in the next paragraph). When this preference is not selected, LabVIEW uses its own platform-independent file dialog boxes, which add some convenient features, such as providing a list of recent paths and reducing the steps necessary to save VIs in VI libraries.

In a few cases, the operating system's native file dialog box does not provide the functionality required by LabVIEW. In such cases, LabVIEW may use its own file dialog box even when this option is selected. Under the Macintosh, if you wire the pattern input or if you wire the datalog type input and use a select mode of 1 (select new file), 2 (select new or existing file), 4 (select new directory), or 5 (select new or existing directory), LabVIEW uses its own file dialog box. Under Windows 3.x, Windows 95, and Windows NT, even if you wire the pattern input, LabVIEW uses its own file dialog box.

(Windows, Macintosh) **Allow drop-through clicks**--Lets you set LabVIEW so that clicking only once on the mouse will activate and select an object in an inactive window. This reduces the number of mouse clicks required from two to one. (Normally, the first click activates the window and the second click passes through as a click in the window.)

Use hot menus--Lets you navigate the menus using the mouse without keeping the left mouse button pressed. This ergonomic feature relieves strain on the hand.

Note: Changes to options in the Miscellaneous Preferences dialog box take effect immediately.

How Preferences are Stored

You do not usually have to edit preference information manually, or know its exact format, because the Preferences dialog box takes care of it for you. Preferences are stored differently on each platform, as described in the following topics.

Preferences Storage in Windows
Preferences Storage on the Macintosh
Preferences Storage in UNIX

Preferences Storage in Windows

Preference information is stored in a `LabVIEW.INI` file in your LabVIEW directory. The format for this file is similar to other `.INI` files, such as the `WIN.INI` file. It begins with a section marker `[LabVIEW]`. This is followed by variable labels and their values, such as `totalMemSize=5000000`. If a configuration value is not defined in `LABVIEW.INI`, LabVIEW checks to see if there is a `[LabVIEW]` section of your `WIN.INI` file, and if so, checks for the configuration value in that file.

You can also specify a preference file on the command line when you start LabVIEW. For example, to use a file named `lvrc` instead of `.labview.ini`, you would type

```
labview -pref lvrc
```

Preferences Storage on the Macintosh

LabVIEW creates a LabVIEW Preferences file in your System Folder that contains preference information. Under System 6, this preference file is created at the top level of the System Folder, while under System 7, it is created in the Preferences folder of the System Folder.

If you want, you can copy your LabVIEW Preferences file to the LabVIEW folder. When launched, LabVIEW always looks for the Preferences file in the LabVIEW folder. If not found there, it looks in the System Folder, and if not found there, it creates a new one in the System Folder. By moving Preferences files between desktop and the LabVIEW folder, you can create multiple Preferences for multiple users or uses.

Preferences Storage in UNIX

Preference information is normally stored in a `.labviewrc` file in your home directory. If you change a parameter from the Preferences dialog box, LabVIEW writes the information to this file. The following information is supplied for those who want to know more about the storage format and the rules for finding preference information.

Preference entries consist of a *preference name* followed by a colon and a value. The preference name is the executable name followed by a period (.) and a token. When LabVIEW searches for preference names, the search is case-sensitive. You can optionally enclose the preference value in double or single quotation marks. For example, to use a default precision of double and use a search path that recursively searches the a particular directory, you could specify the following preference entries.

```
labview.defPrecision : double

labview.viSearchPath : "/usr/lib/labview/*"
```

LabVIEW also searches for preferences in a file named `labviewrc` in the applications directory. For example, if you have installed your LabVIEW files in `/opt/labview`, preferences are read out of both `/opt/labview/labviewrc` and the `.labviewrc` file in your home directory.

Entries in the `.labviewrc` file override conflicting entries in the `labviewrc` file. You may want to use

this global preference file to store things that are the same to all users such as the VI search path.

You can also specify a preference file on the command line when you start LabVIEW. For example, to use a file named `lvrc` instead of `.labviewrc`, you would type

```
labview -pref lvrc
```

Notice that the `labviewrc` file is still read in this case. Also notice that changes made in the Preferences dialog box are written to the `lvrc` file in this example.

Customizing the Controls and Functions Palettes

LabVIEW provides a powerful set of tools for customizing the **Controls** and **Functions** palettes.

These tools are discussed in the following topics:

Adding VIs and Controls to user.lib and instr.lib
Customizing Palettes with the Palettes Editor
Moving Subpalettes
How Views Work

Adding VIs and Controls to user.lib and instr.lib

The simplest method for adding new entries to the **Controls** and **Functions** palettes is to save them inside of the `user.lib` directory. When you restart LabVIEW, the User Libraries subpalette of the **Functions** palette will contain subpalettes for each directory, llb, or `.mnu` file in `user.lib` and entries for each file in `user.lib`.

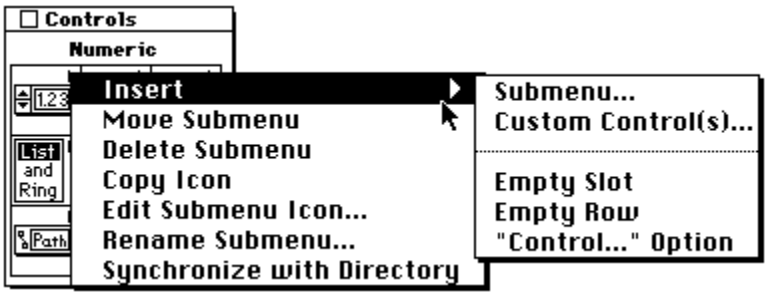
The Instrument I/O palette of the **Functions** palette corresponds to the `instr.lib` directory. You may want to put instrument drivers in this directory to make them easily accessible from the palettes.

Caution: The `vi.lib` directory contains files from National Instruments. You should not save files into `vi.lib` because these files are overridden when you upgrade LabVIEW.

Customizing Palettes with the Palettes Editor

For more control over the layout and contents of the **Controls** and **Functions** palettes, use the **Edit»Edit Control & Function Palettes** option. When you have selected this option, you enter the Palettes Editor. The Edit Palettes dialog box appears.

In this editor, you can rearrange the contents of palettes by dragging objects to new locations. If you want to delete, customize, or insert objects, pop up on a palette as shown in the following illustration, or pop up on an object within a subpalette.



With the pop-up options, you can modify anything within the **User Libraries** menu (corresponds to `user.lib`) or the Instrument I/O menu (corresponds to `instr.lib`). If you want to edit the top-level **Controls** or **Functions** palettes or any of the other predefined menus, you must first create a new view by selecting **new setup...** from the **menu setup** ring in the Edit Palettes dialog box. After selecting a new setup, any menu that you modify that is part of the default menu set will be copied to your view's directory in the menus directory before changes are made to it. This protection of the built-in palettes ensures that you can experiment with the palettes without corrupting the default view.
If you want to add a new object in a new row or column of a subpalette, pop up in the space at the right edge or bottom of the subpalette. You can also create new rows or columns by dragging objects to the area to the right or bottom of the palette.

Installing and Changing Views
Creating Subpalettes

Installing and Changing Views

Controls and **Functions** palette information is stored in the menus directory of your LabVIEW directory. The menus directory contains directories corresponding to each view that you create or install.

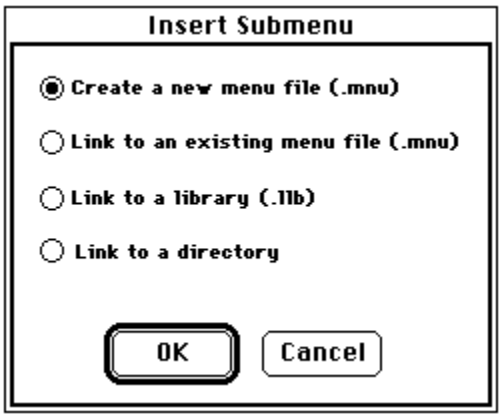
This organization makes it easy to transfer views to other people. To give someone else a copy of a view, give him or her a copy of the view's directory from the menus directory. After placing it in his or her menus directory, the view will become available.

To switch to another view, select **Edit»Edit Control & Function Palettes**. Then select the view that you desire from the **menu setup** ring.

Creating Subpalettes

Once you add a palette, you can move it to a new location, edit the subpalette icon, or rename the palette using the Palettes Editor.

If you want to create a palette from scratch or "hook in" a palette that is not in `user.lib` or `vi.lib`, you can use the **Insert»Submenu...** option from the pop-up menu in the Palettes Editor. When you select this option, you are presented with the following dialog box.



Submenu information can be stored in VI libraries or in .mnu files. A menu file or LLB can contain one **Functions** palette and one **Controls** palette.

Select **Create a new menu file** to insert a new, empty palette. You are then prompted for a name for the palette and a file to contain it. It is recommended that you add a .mnu extension to the file to indicate that it is a menu (palette). It is also recommended that you either store the .mnu file in your view's directory in the menus directory or in the same directory that contains the controls or VIs that the menu largely represents.

Select **Link to an existing menu file** if you have an existing palette that you want to add to the **Controls** or **Functions** palette file.

Select **Link to a directory** if you want to create a palette with entries for all the files in the directory. Selecting the option will also recursively create subpalettes for each of the subdirectories, VI libraries, or .mnu files within the directory. These palettes will automatically update if you add new files to or remove files from the directories that you selected. You can enable or disable the automatic update setting for a subpalette by selecting the **Automatic Update From Directory** pop-up option. This option actually creates menu (palette) files inside of each directory that is recursed into.

Select **Link to a library** if you want to link to the **Controls** palette or **Functions** palette that is a part of a VI library. As with the previous setting, the palette for a library will automatically update as you add files to the library.

Notice that you can mix VIs, functions, and subpalettes within a palette freely. Also, a palette can contain VIs from different locations.

Moving Subpalettes

Because clicking on a subpalette opens it, you cannot move a subpalette by dragging it. To move a subpalette, you can select the **Move Submenu** option from its pop-up menu. As a shortcut, you can hold down the <Shift> key while you click a subpalette to drag it instead of opening it.

How Views Work

Both .mnu files and VI library files are binary files that can contain one **Controls** palette and one **Functions** palette. In addition, both types of files contain an icon for the **Controls** and **Functions** palettes. Each submenu that you create must be stored in a separate file.

When you select a view, LabVIEW looks for a directory in the menus directory. It builds the top level **Controls** and **Functions** palettes from the root.mnu file within that directory. The top level palettes then link to other .mnu or VI library files for each submenu within the palette.

If you link in a directory, LabVIEW looks to see if the directory contains a dir.mnu file. If so, it uses that .mnu file as the submenu for the directory. Otherwise LabVIEW creates a dir.mnu file based on the contents of the directory. For each VI (or control), an entry is created. For each subdirectory, .mnu file, or VI library, a submenu is created.

LabVIEW automatically updates the palettes within a VI library as you add files to or remove files from the VI library. You can optionally set a .mnu file to update based on the contents of a directory. To change this setting, select **Synchronize with Directory** from the submenu icon's pop-up menu. Although it is a good idea to have the .mnu file in the same directory with which it is synchronized, you can actually select any directory. In addition to reflecting a specific directory, you can have additional VIs or controls from other directories in the same palette. You can also remove VIs from the palette by selecting **Delete Item** from the pop-up menu.

Front Panel Object Introduction

This topic introduces the front panel and its two components--controls and indicators. It also explains how to import graphics from other programs to use in your controls.

Building the Front Panel

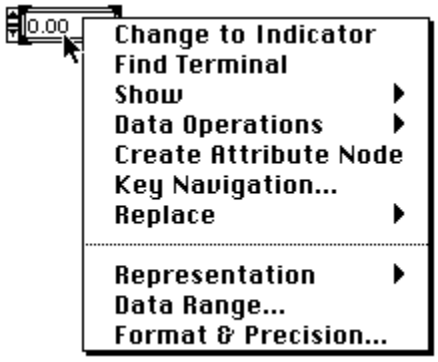
Controls and indicators on the front panel are the interactive input and output terminals of the VI. You use controls to supply data to a VI, and indicators display the data generated by the VI. This section explains a few editing options common to all controls and indicators.

The **Controls** palette on the front panel is shown in the following illustration. Click on one of the icons in the following palette for more information about that topic.

If you idle your cursor over any square of the panel, the name of the collection of controls is displayed.
Front Panel Control and Indicator Common Options
Control Replacement
Key Navigation Dialog Box
Panel Order
Dialog Box Controls

Front Panel Control and Indicator Common Options

When you pop up on a control or indicator on the front panel while editing a VI, you get a menu like the one shown in the following illustration. The options above the line in the pop-up menu are common to all controls and indicators. A few controls and indicators do not have any of the options shown below the line--**Representation**, **Data Range...**, and **Format & Precision...** --in their pop-up menus.



Objects in the **Controls** palette are initially configured as controls or indicators. For example, if you choose a toggle switch from the **Boolean** palette, it appears on the front panel as a control, because a toggle switch is usually an input device. Conversely, if you select an LED, it appears on the front panel as an indicator, because an LED is usually an output device. However, you can reconfigure all controls to be

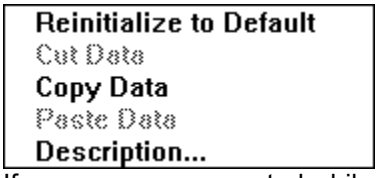
indicators, and vice versa, by choosing the **Change to Control** or **Change to Indicator** commands from the object pop-up menu. The **Numeric** palette contains both a digital control and a digital indicator because you use both frequently. The **String** palette also contains both a string control and a string indicator.
The **Find Terminal** item of the control and indicator pop-up menus highlights the block diagram terminal for the control or indicator. This option is useful for identifying a particular object on a crowded block diagram.

The **Create Attribute Node** item creates an attribute node for the object. Attribute nodes are used to control various properties of the object programmatically.

The **Show** submenu shows a list of the parts of a control that you can choose to hide or show, such as the name label.

When editing a VI, the pop-up menu for a control contains a **Data Operations** submenu. Using items from this menu, you can cut, copy, or paste the contents of the control, set the control to its default value, make the current value of the control its new default, and read or change the control's description. You can copy the data in a control or indicator and also paste this data into another control of the same data type. (When editing a VI, you can paste data into an indicator.) Some of the more complex controls have additional options; for example, the array has options you can use to copy a range of values and to show the last element of the array.

The following illustration shows the edit mode **Data Operations** submenu for a numeric control. This submenu is the only part of a control’s pop-up menu available while the VI is running.



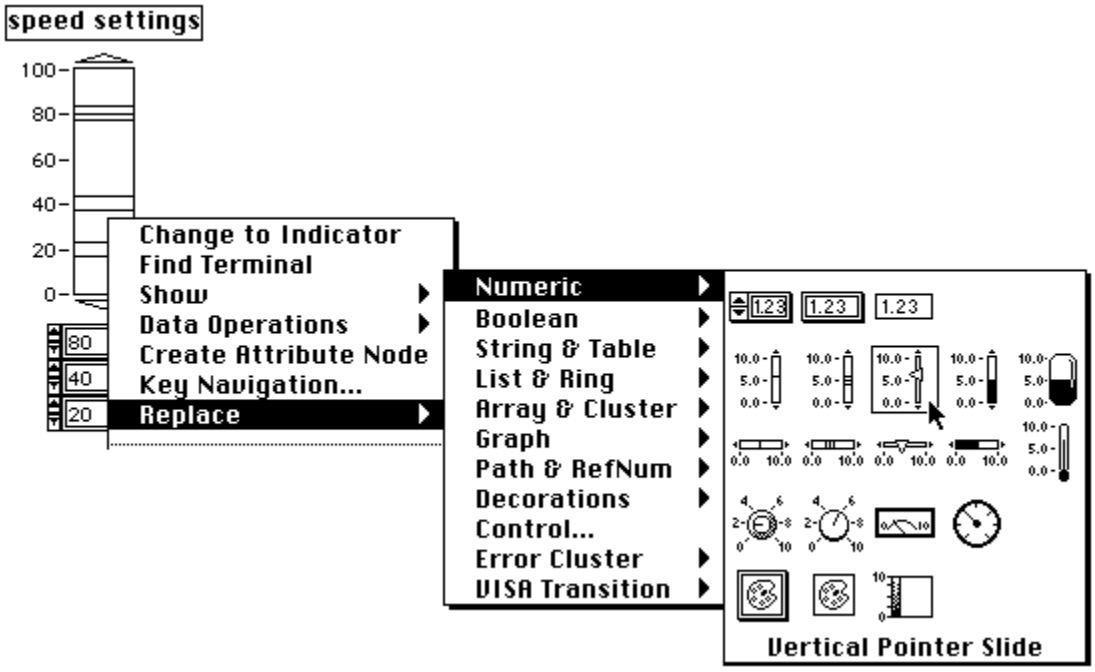
If you pop up on a control while running, you can only change the value of a control. While running, you cannot change most characteristics of a control, such as its default value or description.

Control Replacement

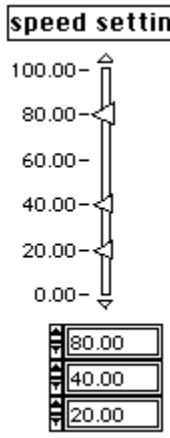
The **Replace** option of an object's pop-up menu displays the **Controls** palette from which you can choose a control or indicator to take the current item's place on the front panel.

Use **Replace** when you want to choose a different style of control, but do not want to lose all of the editing options you have selected for the control up to that point. Selecting **Replace** from the pop-up menu preserves as much information as possible about the original control, such as its name, description, default data, dataflow direction (control or indicator), colors, size, and so on. However, the new control keeps its own data type. Wires from the terminal of the control or local variables remain connected on the block diagram.

The more similar the control is to the one being replaced, the more the original characteristics can be preserved. As an example, the following illustration shows a slide being replaced by a different style of slide.



The next illustration shows the result.



The new slide has the same height, scale, value, name, description, and so on.
If you were to pop up and replace the slide with a string instead, only the name, description, and dataflow direction would be preserved, because a slide does not have much in common with a string.
Another way to replace a control that does not use the **Replace** pop-up menu and does not preserve any characteristics of the old control involves copying the control to the Clipboard. This method does keep the connections on the block diagram and VI connector. Copy the new control to the Clipboard. Then select the old control that you want to replace with the Positioning tool, and select **Edit**»**Paste**. The old control is discarded and the new control is pasted in its place.

Key Navigation Dialog Box

All front panel controls have a **Key Navigation…** option. You use this option to associate a keyboard key combination with a control. When a user enters that key combination while running the VI, LabVIEW acts as though the user had clicked on that control. The associated control becomes the key focus. If the control is a text control, any existing text within that control is highlighted, ready for editing. If the control is a Boolean control, the state of the button is toggled.

The **Key Navigation…** option is disabled for indicators, because you cannot enter data into an indicator.

You can also use the **Key Navigation…** option to specify whether a control should be included when the user tabs from control to control while running.

You can use the **Key Navigation…** option to associate function keys with various buttons that control the behavior of a panel. You can use it to define a default button for VIs that behave like a dialog box, so that pressing the <Enter> (Windows and HP-UX), or <Return> (Macintosh and Sun) key becomes the same as clicking on the default button.

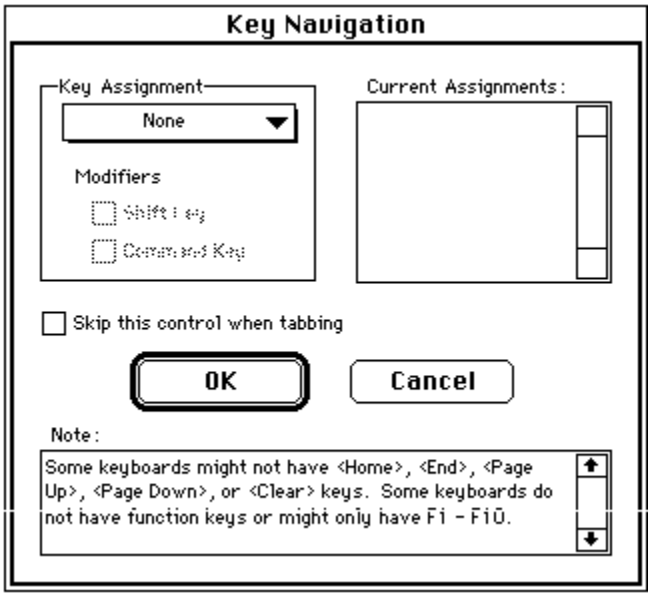
If you associate the <Enter> or the <Return> key with a dialog box button, LabVIEW automatically draws that button with a special thick border around it, as shown in the following illustration.

There are two special things to know about the <Enter> or the <Return> key:

1. If you associate the <Enter> or the <Return> key with a control, then no control on that front panel ever receives a carriage return. Consequently, all strings on that panel will be limited to a single line.
2. When you tab from control to control, buttons are treated specially. If you tab to a button and press the <Return> or the <Enter> key, the button you tabbed to is toggled, even if another control has the <Return> or <Enter> key as its key navigation setting.

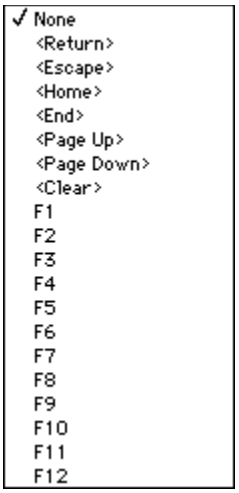
You cannot assign the same key combination to more than one control in a given panel.

When you select the **Key Navigation…** option, the following dialog box is displayed.



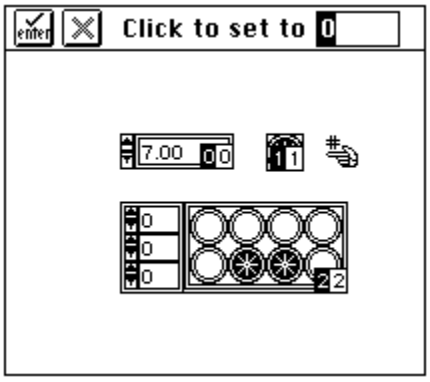
You use the ring at the top left of this dialog box to select the key for this control. You can define the key to be a single key, or a key held in combination with a modifier key, such as the <Shift> key. In addition to the <Shift> key modifier, you can choose the menu key, which is the <Alt> key in Windows, the <command> key on the Macintosh, and the <meta> key in UNIX. The options in this ring are shown in the following illustration.

A list of the current keyboard associations, labeled Current Assignments, is located at the top right of the Key Navigation dialog box (shown in the preceding illustration).



Panel Order

Controls and indicators on a front panel have a logical order, called panel order, that is unrelated to their position on the front panel. The first control or indicator you create on the front panel is element 0, the second is 1, and so on. If you delete a control or indicator, the panel order adjusts automatically. You can change the panel order by selecting **Edit**»**Panel Order…**. The appearance of the front panel changes, because it is now in panel order edit mode, as shown in the following illustration:



The white boxes on the controls and indicators show their current places in the panel order. Black boxes show the new place in the panel order of the control or indicator. Clicking on an element with the panel order cursor sets the elementÕs place in the panel order to the number displayed inside the **Tools** palette. You can change this number by typing a new number into it. When the panel order is the way you want it, click on the enter button to set it and exit the panel order edit mode. Click on the X button to revert to the old panel order and exit the panel order edit mode.
The panel order determines the order in which the controls and indicators appear in the records of datalog files produced by logging the front panel.

Dialog Box Controls

There are several types of dialog box controls: dialog rings (which are numeric), dialog buttons (which are Boolean), dialog checkmarks, and dialog buttons.

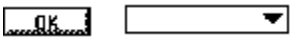
dialog ring

dialog button

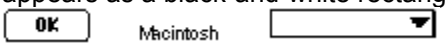
The dialog controls change appearance depending on which platform you are using. Each appears with

the color and appearance typical of that platform.

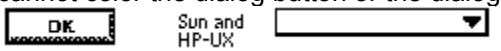
(Windows) The dialog button appears as a three-dimensional rectangular gray button. The dialog ring appears as a flat black-and-white rectangle, as shown in the following illustration.



(Macintosh) The dialog button appears as a two dimensional black-and-white oval. The dialog ring appears as a black-and-white rectangle with a drop shadow, as shown in the following illustration



(UNIX) The dialog button appears as a three-dimensional rectangular gray button. The dialog rings appears as a flat, black-and-white rectangle with drop shadow, as shown in the following illustration. You cannot color the dialog button or the dialog ring.



Because these controls change appearance, you can create VIs with controls whose appearance is compatible with any of the computers that can run LabVIEW. Using these controls, along with the checkmark and radio button Booleans, simple numeric controls, simple strings, and the dialog fonts, you can create a VI that adapts its appearance to match any computer you use the VI on. By using the VI Setup options to hide the menu bar and scroll bars and control the window behavior, you can create VIs that look like standard dialog boxes for that computer.

Importing Graphics from Other Programs

You can import graphics from other programs for use as background pictures, as items in ring controls, or as parts of other controls. (For more information on using graphics in these controls, see the List and Ring Controls and Indicators, and the Custom Controls and Type Definitions topics.)

Before you can use a picture in LabVIEW, you have to load it into the LabVIEW Clipboard. There are one or two ways to do this, depending on your platform, as described below.

(Windows) If you can copy an image directly from a paint program to the Windows Clipboard and then switch to LabVIEW, LabVIEW automatically imports the picture to the LabVIEW Clipboard. Or you can use the **Import Picture from File...** option from the Windows **Edit** menu to import a graphics file into the LabVIEW Clipboard. In Windows 3.1, you can use the latter method on CLP, EMF, GIF, PCX, BMP, TARGA, TIFF, LZW, WMF, and WPG files. On Windows 95 and Windows NT, you can use EMF, WMF, and CLP files.
(Macintosh) If you copy from a paint program to the Clipboard and then switch to LabVIEW, LabVIEW automatically imports the picture to the LabVIEW Clipboard.

(UNIX) You can use the **Import Picture from File...** option from the UNIX **Edit** menu to import a picture of type X Window Dump (XWD), which you can create using the xwd command.

(All Platforms) Once a picture is on the LabVIEW Clipboard, you can paste it as a static picture on your front panel, or you can use the **Import Picture** option of a pop-up menu, or the **Import Picture** options in the Control Editor.

For an example of how to import graphics from other programs, see `examples\general\controls\custom.llb`.

Numeric Controls and Indicators

Numeric controls–for entering and displaying numeric quantities.

Boolean Controls

Boolean controls–for entering and displaying True/False values.

String and Table Controls

String & Table controls–for entering and displaying text.

List and Ring Controls

List & Ring controls–for displaying and/or selecting from a list of options.

Array and Cluster Controls

Array and Cluster controls–for grouping sets of data.

Graph Controls

Graph controls–for plotting numeric data in chart or graph form.

Path and Refnum Controls

Path & Refnum controls–for entering and displaying file paths and for passing refnums from one VI to another.

Decorations

Decorations–for adding graphics to customize front panels. These objects are for decoration only, and do not display data.

Select a Control...

Select a Control...–for choosing a custom control of your own design.

LabVIEW Function Error Codes

Code	Description
0	No error.
1	Manager argument error.
2	Argument error.
3	Out of zone.
4	End of file.
5	File already open.
6	Generic file I/O error.
7	File not found.
8	File permission error.
9	Disk full.
10	Duplicate path.
11	Too many files open.
12	System feature not enabled.
13	Resource file not found.
14	Cannot add resource
15	Resource not found.
16	Image not found.
17	Image memory error.
18	Pen does not exist.
19	Config type invalid.
20	Config token not found.
21	Config parse error.
22	Config memory error.
23	Bad external code format.
24	Bad external code offset.
25	External code not present.
26	Null window.
27	Destroy window error.
28	Null menu.
29	Print aborted.
30	Bad print record.
31	Print driver error.
32	Windows error during printing.
33	Memory error during printing.
34	Print dialog error
35	Generic print error.
36	Invalid device refnum.
37	Device not found.
38	Device parameter error.
39	Device unit error.
40	Cannot open device.
41	Device call aborted.
42	Generic error.
43	Cancelled by user.
44	Object ID too low.
45	Object ID too high.
46	Object not in heap.
47	Unknown heap.
48	Unknown object (invalid DefProc).
49	Unknown object (DefProc not in table).
50	Message out of range.
51	Invalid (null) method.

52	Unknown message.
53	Manager call not supported.
54	Bad address.
55	Connection in progress.
56	Connection timed out.
57	Connection is already in progress.
58	Network attribute not supported.
59	Network error.
60	Address in use.
61	System out of memory.
62	Connection aborted.
63	Connection refused.
64	Not connected.
65	Already connected.
66	Connection closed.
67	Initialization error (interapplication manager)
68	Bad occurrence.
69	Wait on unbound occurrence handler.
70	Occurrence queue overflow.
71	Datalog type conflict.
72	Unused.
73	Unrecognized type (interapplication manager).
74	Memory corrupt.
75	Failed to make temporary DLL.
76	Old CIN version.
77	Unknown error code
81	Format specifier type mismatch
82	Unknown format specifier
83	Too few format specifiers
84	Too many format specifiers
85	Scan failed

Data Acquisition Common Questions

All Platforms

Where is the best place to get up to speed quickly with data acquisition and LabVIEW?

What is the easiest way to address my AMUX-64T board with my MIO board?

What are the advantages/disadvantages of reading AI Read's backlog rather than a fixed amount of data?

How can I tell when a continuous data acquisition operation does not have enough buffer capacity?

I want to group two or more ports using my DIO32F, DIO24, or DIO-96 board, but I do not want to use handshaking. I just want to read one group of ports just once. How can I setup my software?

I want to use the OUT1, OUT 2, OUT3 and IN1, IN2, IN3 pins on my DIO-32F board. How do I address those pins using the Easy I/O

I want to use a TTL digital trigger pulse to start data acquisition on my DAQ device. I noticed there are two types of triggers: Digital Trigger A, and Digital Trigger A&B. Which digital trigger setting should I use and where should I connect the signal?

When are the data acquisition boards initialized?

Windows Only

I open a VI that calls a DAQ VI, or drop a DAQ subVI on a block diagram, and crash.

I am having problems running Windows for Workgroups with my data acquisition program. (Windows 3.1)

While performing analog input, I get memory allocation errors (-10444) even though I have a large amount of memory on my machine.

I am having problems running Windows for Workgroups with my data acquisition program. (Windows 3.1)

I bought LabVIEW for Windows and also have a slightly older DAQ device from National Instruments. I installed the entire LabVIEW package, but should I go ahead and install my NI-DAQ for Windows drivers that I originally got with the board? (Windows 3.1)

Macintosh Only

My analog input VIs returns error -10845 (buffer overflow). What is the problem?

Where is the best place to get up to speed quickly with data acquisition and LabVIEW?

Read the LabVIEW Data Acquisition Basics Manual and look at the `run_me.llb` examples (in `examples/daq`) included with the package.

What is the easiest way to address my AMUX-64T board with my MIO board?

Set the number of AMUX boards used in the configuration utility (`wdaqconf.exe` on Windows or NI-DAQ control panel on Macintosh). Then in the channel string inputs specify the onboard channel. For example, with one AMUX-64T board, the channel string 0:1 will acquire data from AMUX channels 0 through 7, and so on.

What are the advantages/disadvantages of reading AI Read's backlog rather than a fixed amount of data?

Reading the backlog is guaranteed not to cause a synchronous wait for the data to arrive. However, it adds more delay until the data is processed (because the data was really available on the last call) and it can require constant reallocation or size adjustments of the data acquisition read buffer in LabVIEW.

How can I tell when a continuous data acquisition operation does not have enough buffer capacity?

The scan backlog rises with time, either steadily or in jumps, or takes a long time to drop to normal after an interrupting activity like mouse movement. If you can open another VI during the operation without receiving an overrun error you should have adequate buffer capacity.

I want to group two or more ports using my DIO32F, DIO24, or DIO-96 board, but I do not want to use handshaking. I just want to read one group of ports just once. How can I set it up in software?

Use Easy I/O VIs (Write to Digital Port or Read from Digital Port) or Advanced Digital VIs (DIO Port Config, DIO Port Write or DIO Port Read), and set multiple ports in the port list. For Easy I/O VIs, you can specify up to four ports in the port list. Whatever data you try to output to each port of your group will correspond to each element of the data array. This also applies for input.

I want to use the OUT1, OUT 2, OUT3 and IN1, IN2, IN3 pins on my DIO-32F board. How do I address those pins using the Easy I/O Digital VIs in LabVIEW?

These output and inputs pins are addressed together as port 4. OUT1 and IN1 are referred to as bit 0, OUT2 and IN2 are referred to as bit 1, and OUT3 and IN3 are referred to as bit 2. Only the NB-DIO-32F has 3 pins for each direction. If you use the Write To Digital Port VI, you will output on the OUT pins, and if you use the Read From Digital Port VI, you will input from the IN pins.

I want to use a TTL digital trigger pulse to start data acquisition on my DAQ device. I noticed there are two types of triggers: Digital Trigger A, and Digital Trigger A&B. Which digital trigger setting should I use and where should I connect the signal?

You should use Digital Trigger A, which stands for first trigger," to start a data acquisition. Digital Trigger B, which stands for second trigger," should only be used if you are doing both a start AND a stop trigger for your data acquisition. Connect your trigger signal to either STARTTRIG* (pin 38) if you are using an AT-MIO-16, AT-MIO-16D, NB-MIO-16X, or EXTTRIG* or DTRIG for any other board that has that pin. The only analog input boards on which you cannot do a digital trigger are the PC-LPM-16, DAQCard-700, and the DAQCard-500. Refer to the AITrigger Config description for more information on the use of digital triggers on your DAQ device.

Note: The NB-MIO-16 has an EXTTRIG* pin, but cannot support start and stop triggering.

When are the data acquisition boards initialized?

All data acquisition boards are initialized automatically when the first DAQ VI is loaded in on a diagram when you start LabVIEW. You can also initialize a particular board by calling the Device Reset VI.

I open a VI that calls a DAQ VI, or drop a DAQ subVI on a block

diagram, and crash.

The first time a DAQ VI is loaded into memory in LabVIEW, LabVIEW opens the library (dll) that controls data acquisition. A crash at this time indicates a problem in communicating with the driver. This may indicate that there is a conflict with another device in the machine.

To determine the source of the problem, quit LabVIEW and Windows, re-launch Windows, and run wdaqconf.exe. Run a simple configuration test with the NI boards in the machine. If this results in a crash, there is probably a conflict with another device in the machine or the drivers file versions do not correspond for some reason. If not, you need to obtain the latest version of the DAQ driver from NI BBS, World Wide Web, or FTP site.

We have also seen cases where the video driver conflicts with both WDAQCONF and LabVIEW. You can obtain the *Error Messages and Crashes Common Questions* document from the NI FaxBack system.

I am having problems accessing ports 2 or higher on the AT-MIO-16D or PC-DIO-96.

A problem was found in version 3.0 with addressing the higher-numbered ports on these boards. To fix the problem, get the updated version of `atwdaq.dll`, and use the updated DIO Port Read.vi. These updated files are included with LabVIEW for Windows version 3.0.1. LabVIEW for Windows version 3.0 users can obtain these files by downloading the file `win30up2.zip` from the NI BBS or FTP sites. The file is in the directory `support/labview/windows/LVWin3.0/updates`.

While performing analog input, I get memory allocation errors (-10444) even though I have a large amount of memory on my machine.

Buffers for data acquisition arrays, unlike arrays for LabVIEW buffers, must be available in physical RAM, not in virtual memory. For example, assume the machine has 32 MB of RAM, and LabVIEW is allocated 8 MB of RAM. The memory allocated to LabVIEW wires will come out of the 8 MB; however, data acquisition buffers will be allocated out of the remaining 24 MB of RAM.

On an AT style machine (ISA) using DMA, the DMA controller can only address the first 16 MB of RAM. If you get "out of memory" errors on your machine, try setting the board to use programmed I/O (interrupts only) with the `Set Device Information.vi` (in `DAQ/Calibration and Config`). An alternative is to switch to an EISA bus machine. The DMA controller can address up to 4 GB of RAM on EISA machines.

I am having problems running Windows for Workgroups with my data acquisition program. (Windows 3.1)

Remark out nivisrd.386 in the [386 Enhanced] section of your system.ini file. To mark out the line, place a number sign (#) at the beginning of the line which reads:

```
device = c:\windows\system\nivisrd.386
```

nivisrd.386 normally improves performance by reducing interrupt latencies in Windows enhanced mode.

I bought LabVIEW for Windows and also have a slightly older DAQ device from National Instruments. I installed the entire LabVIEW package, but should I go ahead and install my NI-DAQ for Windows drivers that I originally got with the board? (Windows 3.1)

In most cases, the answer is no. The LabVIEW installer installs a set of DAQ driver files that are guaranteed to work with LabVIEW, whereas if you happen to install an older version of the drivers, you may run into many problems. You may even end up crashing your computer every time you do any data acquisition. If you buy a new DAQ device and if you already have LabVIEW installed, it is safe to install the NI-DAQ for Windows drivers from those disks. In any case, make sure you install and use the latest version of the NI-DAQ drivers.

My analog input VIs returns error -10845 (buffer overflow). What is the problem? (Macintosh)

If you do not have an NB-DMA-2800 or NB-DMA8G board in your Macintosh and are trying to acquire at sampling rates (not scan rates) greater than 8 kHz, you may get this error. Even at sampling rates under 8 kHz, depending on the type of machine, you may run into overflow error problems if there is a lot of other interrupts that need to be serviced. This is all due to the long interrupt latencies. If you do have either DMA board in your Macintosh, make sure that you have a RTSI cable connecting your DAQ device and the DMA board. Even after you connect a RTSI cable, restarting LabVIEW may help.

Also, if you have a Quadra, your errors may be caused by prolonged network interrupt latencies, which prevents the NI-DAQ driver from copying the data in the DAQ device resident memory to the memory on your computer. In this case, you can either disable AppleTalk in the Chooser and disconnect your AppleTalk cable or contact National Instruments and ask for the newest revision of the NB-MIO-16X (which has a larger device resident memory) if you are using either of those DAQ devices.

Communications Common Questions

All Platforms

How do I use LabVIEW to communicate with other applications?
How do I launch another application with LabVIEW?
When would I want to use UDP instead of TCP?
What port numbers can I use with TCP and UDP?
Why cant I broadcast using UDP?

Windows Only

What winsock.dll can I use with LabVIEW?
How do I call an Excel macro using DDE?
Why doesn't DDE Poke work with Microsoft Access?
What commands do I use to communicate with a non-LabVIEW application using DDE?
How do I install LabVIEW as a shared application on a file server?
Why does the Synch DDE Client / Server hang on NT after many transfers?
Are there plans for LabVIEW to support OLE?

Macintosh Only

What is a target ID?
Why can't I see my application in the dialog box generated by PPC Browser?
How can I close the Finder using Apple Events?

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.

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 the 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(s) 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 (e.g. ftp). When you specify a local port, you can use the value of 0 which would cause TCP and UDP to choose an unused port.

Why cant I broadcast using UDP?

Because the broadcast address varies among domains, you need to find out from your system administrator what broadcast address to use. For example, the broadcast address 0xFFFFFFFF is not correct for your domain. Additionally, your machine may default to not allow broadcasting unless the process is run by the root user.

What winsock.dll can I use with LabVIEW?

This question pertains to Windows 3.x only, as Windows 95 and Windows NT include this file in their operating systems.

Any WinSock driver that conforms to standard 1.1 should work with LabVIEW. You can find Information regarding National InstrumentsÕ in-house testing of the `winsock.dll` in your online Release Notes.

Recommended:

- TCPOpen version 1.2.2 from Lanera Corporation (408) 956-8344.
- Trumpet (version 1.0 tested). Available via anonymous ftp to `ftp.utas.edu.au` in the directory `/pc/trumpet/winsock/*`. For information send electronic mail `to trumpet-info@petros.psychol.utas.eduau`.
- Super-TCP version 3.0 R1 from Frontier Technologies Corporation (414) 241-4555.
- NEWT/Chameleon version 3.11 from NetManage, Inc. (408) 973-7171.
- Windows for Workgroups `winsock.dll` from Microsoft.

Not Recommended:

National InstrumentsÕ limited testing of these products yielded various problems and crashes while attempting TCP/IP communication. At this time, National Instruments 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 (408) 741-0781.
- PCTCP version 2.x from FTP Software, Inc. (508) 685-4000.

How do I call an Excel macro using DDE?

Use the 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 UserÕs 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.

What 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 specific application to see which commands are available.

How do I install LabVIEW as a shared application on a file server?

Provided the user has a license for each client, the process is as follows:

- 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 the 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
  ```

  Note: pref must be lower case. Additionally, each local machine must have its own LabVIEW temporary directory. This is done in LabVIEW by choosing EditÈPreferences....

- You do not need GPIB.DLL on the server machine, unless you are using a GPIB board on this machine. You then need the `gpibdrv` file in the LabVIEW directory. Then, on each machine that has

a GPIB board, you need to install the driver for that board. You can do this by either using the drivers that came with the board, or by doing a custom LabVIEW installation, in which only the desired GPIB driver is installed on the local machine.

- The same procedure for GPIB.DLL applies to NI-DAQ.

Why does the Synch DDE Client / Server hang on NT after many transfers?

There are some problems with DDE in LabVIEW for NT that result in VIs hanging during DDE Poke and DDE Request operations. This limitation is specific to Windows NT.

Are there plans for LabVIEW to support OLE?

OLE (Object Linking and Embedding) is a way of embedding objects from one application into another application. For example, a spreadsheet might be included on a word processing document. When the text document is loaded, the current values that are found in the spreadsheet are automatically included into the document. National Instruments is currently investigating support for OLE for a future version of LabVIEW; however, no dates have been set on when a version including OLE support will be available.

OLE Automation is a technique by which Automation servers can expose methods and properties to other applications and Automation controllers can access the methods and properties of other applications. LabVIEW 4.x is an OLE Automation controller. There is a library of VIs, which you can use to execute properties and methods exposed by Automation servers.

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. The target ID to an application can be accessed by one of the following commands:

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 you can use to select an application, which may be across the network or on your computer.

The target ID you generated at the beginning of your VI should be used 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 cannot be used with Apple Events, it does not show up in the PPC Browser dialog box. If you are certain that the desired application supports Apple Events, make sure that you have turned on File Sharing on your Macintosh. Select **Control » Sharing Setup** to turn File Sharing 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.

I get timeout errors with GPIB Read/Write in LabVIEW.

Why can I communicate with my GPIB instrument with a LabVIEW VI running in execution highlighting mode mode but not when it is running full-speed?

Why can I write successfully to my GPIB instrument but can't read back from it?

A VI which talks with a GPIB instrument works fine on one platform and not on another.

Windows Only

I can communicate with my instrument using a Quick Basic program but not from LabVIEW.

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`	Five bytes were sent. Read up to 100 bytes from the instrument.
`dev1: ibrd 100`	

`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 which 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.

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`.

Version 3.0.1 does not surrender time to other applications in this situation and thus greatly improves the performance of GPIB.

Serial I/O

All Platforms

Why doesnt my instrument respond to commands sent with Serial Port Write.vi?
How do I close a serial port so that other applications can use it?
How do I reset or clear a serial port?
How can I add additional serial ports to my computer?
How can I control the DTR and RTS serial lines?
Why can't I allocate a serial buffer larger than 32kbytes?

Windows Only

How do I access the parallel port?
What do the error numbers received from the serial port VIs mean?

Sun Only

I receive an error -37 when performing serial I/O.
Serial I/O hangs on a Solaris 1.x machine.

Why doesnt my instrument respond to commands sent with 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` (line feed). To insert a carriage return, use Concatenate Strings and append a Carriage Return constant to the string, or manually enter (`\r`) after selecting **'\' Codes Display** from the string pop-up menu. To learn more about **'\' Codes Display** , see the Backslash Codes Display Option topic.

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 reset or clear a serial port?

Use `Serial Port Init.vi` to reinitialize the port. This VI will automatically clear the serial port buffers associated with the port number.

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
U.S. Office:	Tel: 512-794-0100 Fax: 512-794-8411
FaxBack:	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 only eight serial ports. Under Windows 95 and Windows NT, LabVIEW can address as many as 256 serial ports if you want. 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:

```
labview.serialDevices=COM1; COM2; COM3; COM4; COM5; COM6; COM7; COM8; COM9;
COM10; LPT1; LPT2; LPT3; LPT4;
```

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 4.0 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**

Sun and HP-UX

When you use the serial port VIs on a Sun SPARCstation under Solaris 1, the port number parameter is 0 for `/dev/ttya`, 1 for `/dev/ttyb`, 2 for `/dev/ttyc`, 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.

Because other vendors 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 version 3.x, a configuration option exists to tell LabVIEW how to address the serial ports. LabVIEW will support any board which uses

standard UNIX devices. Some manufacturers suggest using cua rather than tty device nodes with their boards. LabVIEW for Sun can address both types of nodes.

The file `.labviewrc` (`.Xdefaults` in previous versions)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, 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 *Steps for Creating a CIN* topic in Chapter 1 of the *LABView Code Interface Reference Manual* 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 32kbytes 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.

How do I access the parallel port?

In LabVIEW for Windows, port 10 is LPT1, port 11 is LPT2, and so on. To send data to a printer connected to a parallel port, use the `Serial Port Write.vi`. See the Serial Port VIs, topic for more details.

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	Meaning
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.
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.
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).

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 Configuration Options section in the *LabVIEW for Sun 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.