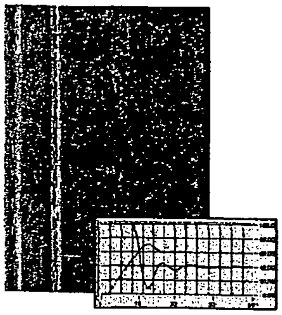
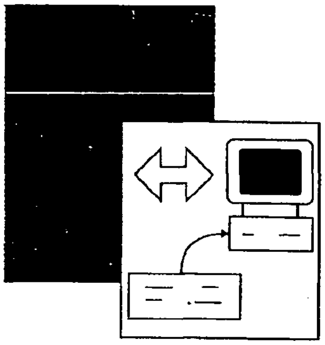
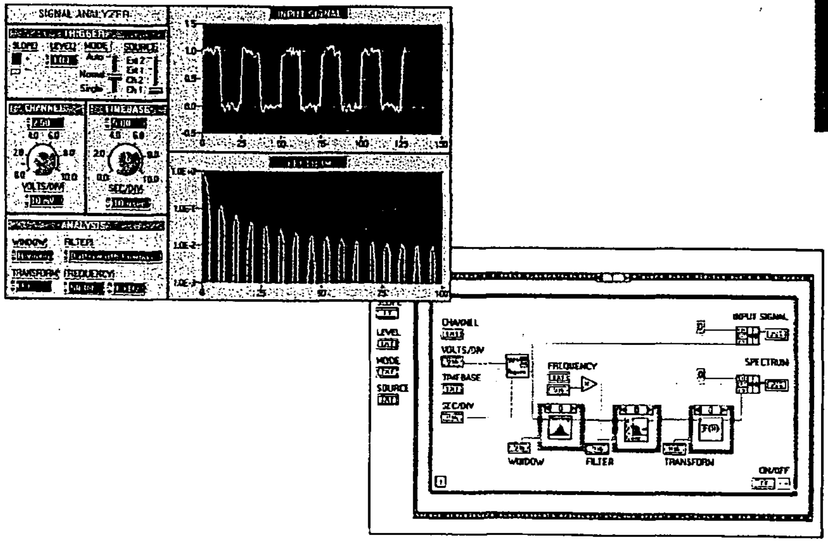
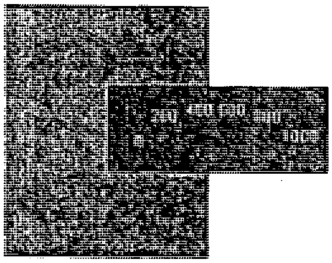






LabVIEW® for Windows

User Manual

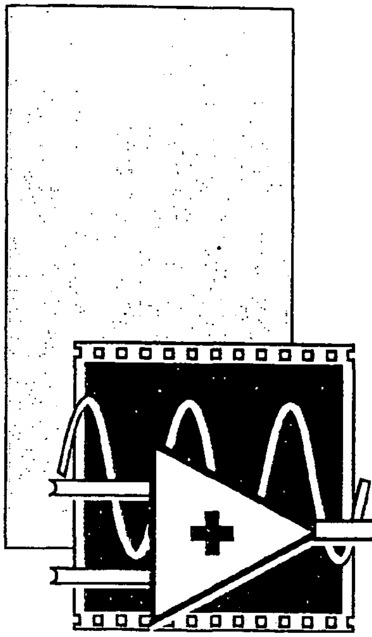


NATIONAL INSTRUMENTS®
The Software is the Instrument®

August 1993 Edition
Part Number 320534-01

ABB Inc.

EXHIBIT 1014



LabVIEW® for Windows

User Manual

Part Number 320534-01
August 1993

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
(800) 433-3488 (toll-free U.S. and Canada)
Technical support fax: (512) 794-5678

Branch Offices:
Australia 03 879 9422, Austria 0662 435986, Belgium 02 757 00 20, Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 65 33 70, Germany 089 714 50 93, Italy 02 48301892, Japan 03 3788 1921, Netherlands 01720 45761, Norway 03 846866, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 27 00 20, U.K. 0635 523545

Limited Warranty

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

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

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

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

Copyright

Trademarks

Warning Regarding Medical and Clinical Use of National Instruments Products

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

Contents

Contents

Contents

LabVIEW Basics—Chapters 1 through 3

Front Panel Objects—Chapters 4 through 10

Chapter 5
Numeric Controls and Indicators

Chapter 6
Boolean Controls and Indicators

Chapter 7
String and Table Controls and Indicators

Chapter 8
Array and Cluster Controls and Indicators

Chapter 9
Graph and Chart Indicators

Block Diagram Programming—Chapters 11 through 19

Appendix B

Appendix C

Figure

Tables

About This Manual

About This Manual

Introduction

About This Manual

The *LabVIEW for Windows User Manual* discusses how to create, edit, and execute virtual instruments (VIs) using LabVIEW. This manual explains the front panel and block diagram; numeric, Boolean, string, array and cluster controls and indicators; graphs and charts; and wiring the block diagram.

You should read the LabVIEW release notes and your tutorial manual before you use the *LabVIEW for Windows User Manual.*

Organization of This Manual

This manual covers three subject areas. Chapters 1 through 3 introduce basic LabVIEW ideas. Chapters 4 through10 explain how to use front panel objects. Chapters 11 through 19 explain block diagram programming objects and techniques.

LabVIEW Basics

- Chapter 1, *Introduction to LabVIEW,* discusses the unique LabVIEW approach to programming. It also contains basic information that explains how to start using LabVIEW to develop programs, referring you to other chapters or manuals for more information.
- Chapter 2, *Editing VIs*, discusses editing techniques for the front panel and the block diagram.
- Chapter 3, *Executing VIs*, discusses how to operate and debug VIs, explains how to set up VIs and subVIs for special execution modes, and discusses factors that affect execution speed.

Front Panel Objects

- Chapter 4, *Introduction to Front Panel Objects*, introduces the front panel and its two component parts, controls and indicators.
- Chapter 5, *Numeric Controls and Indicators*, explains how to edit and operate numeric controls and indicators.
- Chapter 6, *Boolean Controls and Indicators*, discusses how to create and operate Boolean controls and indicators.
- Chapter 7, *String and Table Controls and Indicators*, discusses how to use string controls and indicators, and the table.
- Chapter 8, *Array and Cluster Controls and Indicators*, describes how to use LabVIEW arrays and clusters.
- Chapter 9, *Graph and Chart Indicators*, describes how to create and use graph and chart indicators.
- Chapter 10, *Path Controls and Refnums*, describes how to use file path controls and refnums.

Block Diagram Programming

- Chapter 11, *Introduction to the Block Diagram*, describes nodes, terminals, and wires–the elements you use to build a block diagram.
- Chapter 12, *Wiring the Block Diagram*, explains how to wire the block diagram, which is the VI program, and how to debug nonexecutable VIs.
- Chapter 13, *Structures*, describes how to use the For Loop, While Loop, Case, and Sequence structures.
- Chapter 14, *The Formula Node*, describes how to use the Formula Node to execute mathematical formulas on the block diagram.
- Chapter 15, *Attribute Nodes*, describes how to use Attribute Nodes to set and read attributes of front panel controls programmatically
- Chapter 16, *Global and Local Variables*, describes how to define and use global and local variables.
- Chapter 17, *File VIs*, describes the utility VIs that perform high- and intermediate-level file I/O operations.

- Chapter 18, *Error Handler* VIs, describes a set of VIs for managing and reporting errors.
- Chapter 19, *VI Setup Options*, lists the VI setup options you can use to create special effects in your VIs.

Appendices, Glossary, and Index.

- Appendix A, *Data Storage Formats*, discusses the formats in which LabVIEW saves data.
- Appendix B, *Error Codes*, contains tables of numerical error codes drawn from the entire LabVIEW documentation set.
- Appendix C, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes parameter names, menus, menu items, or dialog box buttons or options.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Monospace font denotes text or characters that you enter using the keyboard. File names, directory names, drive names, sections of code, programming examples, syntax examples, and messages and responses that the computer automatically prints to the screen also appear in this font.

<> Angle brackets enclose the name of a key on the keyboard–for example, <f>.

- A hyphen between two or more key names enclosed in angle brackets denotes that you hold down the first key while you press the next key(s)–for example, <shift-enter>.

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

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

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

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

Customer Communication

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

LabVIEW Basics
Chapters 1 through 3

LabVIEW Basics

Chapters 1 through 3

LabVIEW Basics **Chapters 1 through 3**

Chapter 1

Introduction to LabVIEW

This chapter discusses the unique LabVIEW approach to programming. It also contains basic information that explains how to start using LabVIEW to develop programs. The chapter refers you to other chapters or manuals for more information. Please read the LabVIEW release notes and your tutorial manual before you use the *LabVIEW for Windows User Manual.*

What Is LabVIEW?

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

LabVIEW, like C or BASIC, is a general-purpose programming system with extensive libraries of functions and subroutines for any programming task. LabVIEW also contains application-specific 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 imitate actual instruments. However, they are identical to functions from conventional language programs. VIs have an interactive user interface, a source code equivalent, and accept parameters from higher level VIs. These three VI features are discussed below.

- 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 input data using a mouse and keyboard, and then view the results on the computer screen.
- The VI receives instructions from a *block diagram*, which you construct in G. The block diagram is a pictorial solution to a programming problem. The block diagram is also the source code for the VI.
- VIs are hierarchical and modular. You can use them as top-level programs, or as subprograms within other programs or subprograms. A VI within another VI is called a *subVI*. The *icon and connector* of a VI work like a graphical parameter list so that other VIs can pass data to a subVI.

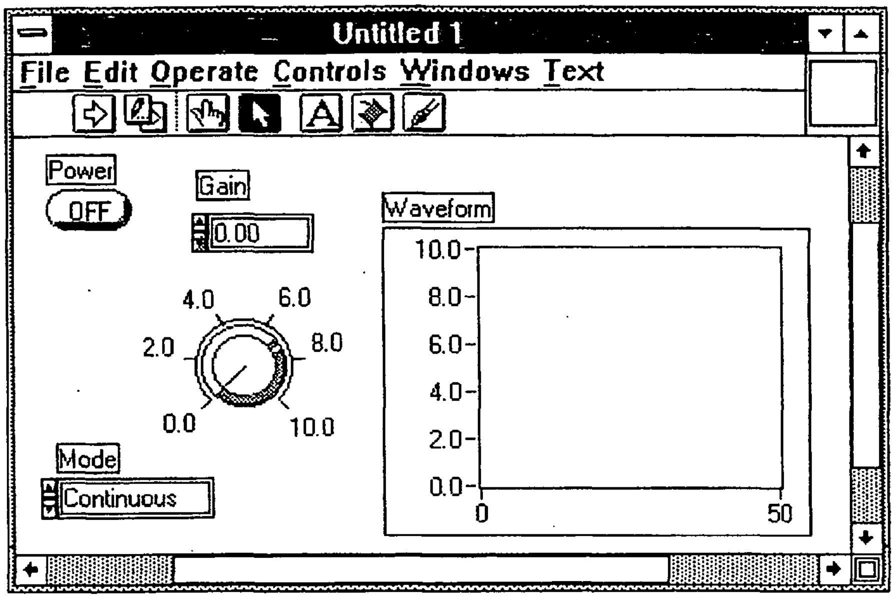
With these features, LabVIEW promotes and adheres to the concept of *modular programming*. You divide an application into a series of tasks, which you can divide again until a complicated application becomes a series of simple subtasks. You build a VI to accomplish each subtask and then combine those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

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

The next sections further explain the front panel, block diagram, icon, connector, and other related features.

Front Panel

The user interface of a VI is the same as the user interface of a physical instrument–the front panel. A front panel of a VI might look like the following illustration.



The Panel window displays the front panel of a VI. The front panel 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.

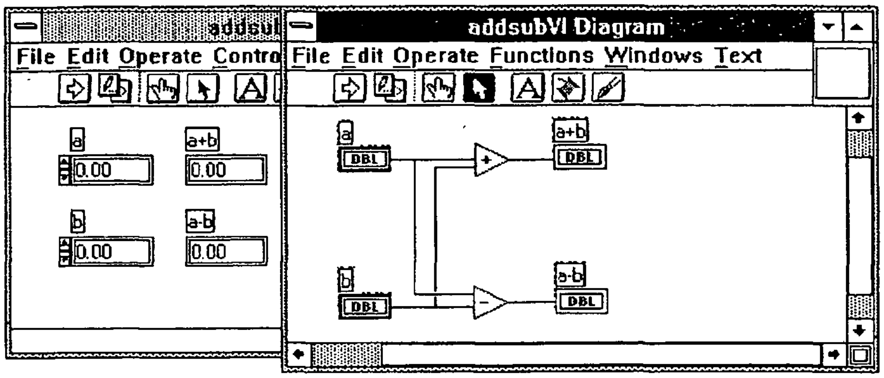
You add controls and indicators to the front panel by selecting them from the **Controls** menu at the top of the Panel window. You can change the size, shape, and position of an object. In addition, each object contains a pop-up menu with which you can change object attributes. You access this pop-up menu by clicking with the right hand mouse button on the object.

Read Chapter 2, *Editing VIs*, then see the *Front Panel Objects* section of this manual (Chapters 4 through 10) for information on building a front panel.

Block Diagram

The Diagram window holds the block diagram of the VI, 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, similar to loops and case statements in traditional programming languages, repeatedly or conditionally execute code. 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 terminals. You cannot wire a source terminal to another source or a sink terminal to another sink, but 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 data arrives at all its input terminals; the node supplies data to all of its output terminals when it finishes executing; and the data passes immediately from source to 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 the Block Diagram Programming part of this manual (Chapters 11 through 19) for in-depth information on using block diagram objects to build a program.

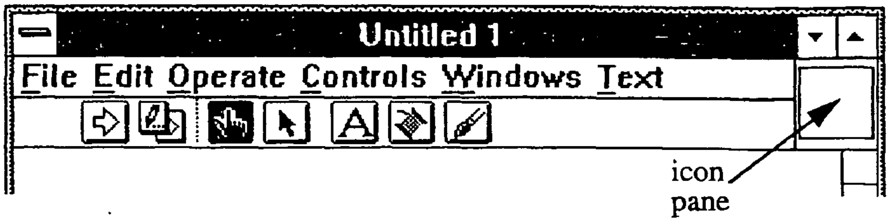
Icon and Connector

When your VI operates as a subVI (the LabVIEW analog of a subroutine) inside another VI, the controls and indicators receive data from and return data to the calling VI.

The *icon* represents a VI in the block diagram of another VI. The *connector* is a set of terminals that correspond to the subVI controls and indicators. The icon can be the pictorial representation of the purpose of the VI, or it can be 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 Panel and Diagram windows. The default icon is a blank frame, as shown in the following illustration.



Every VI also has a connector, which you access by choosing **Show Connector** from the Panel icon pane pop-up menu. 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 20 terminals. You can leave terminals unconnected, then connect them later without invalidating old VIs.

These topics are more fully discussed in Chapter 2, *Editing VIs*.

Introduction to the LabVIEW Development System

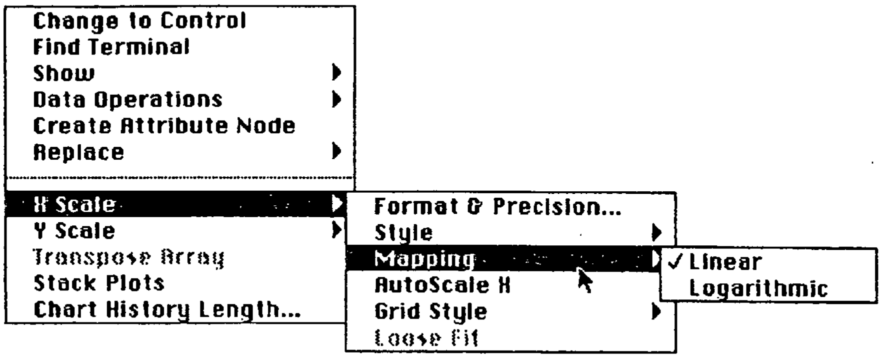
This section explains basic features of the LabVIEW development system, such as menus, tools, windows and modes, and explains how to load, print, and save VIs. Subsequent chapters explain how to use LabVIEW VI editing and operating features.

Using LabVIEW Menus

LabVIEW uses menus extensively. The *menu bar* at the top of a VI window 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 many applications, such as **Open**, **Save**, **Copy**, and **Paste**, and many others particular to LabVIEW. Many menus also list shortcut key combinations for you to use if you choose.

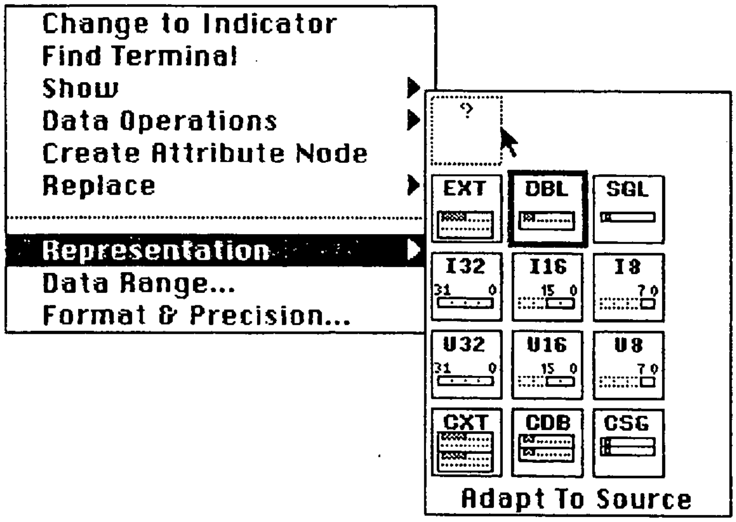
The LabVIEW menu you will use most often is the object *pop-up menu*. Virtually every LabVIEW object, as well as empty front panel and block diagram space, has a pop-up menu of options and commands. Instructions throughout the manual suggest that you select a command or option from an object pop-up menu. Your LabVIEW tutorial manual explains popping up in more detail.

Menu items that expand into submenus are *hierarchical menus* and are denoted by a right arrowhead, as shown in the following illustration.



Hierarchical menus sometimes have a selection of mutually exclusive options. The currently selected option is denoted by a check mark, as with the **Linear** option in the preceding illustration.

The **Representation** menu shown in the following illustration is an example of a *palette menu*, which uses icons to represent options. Notice that the currently selected option in a palette menu is framed or highlighted.



Menu items leading to *dialog boxes* are denoted by ellipses, as in **Data Range...** above.

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

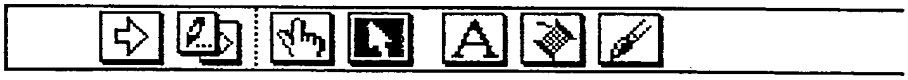
Diagram and Panel Windows

Each VI has two separate but related windows. The *Panel* window contains the front panel of your VI. The *Diagram* is the window in which you build your block diagram. You can switch between windows with the **Show Panel/Show Diagram** command in the **Windows** menu. You can also position them side-by-side with the **Tile** command, also in the **Windows** menu.

Edit and Run Modes

A VI is either in *edit mode*, in which you can create or change a VI, or *run mode*, in which you can execute a VI. You cannot edit a VI in run mode.

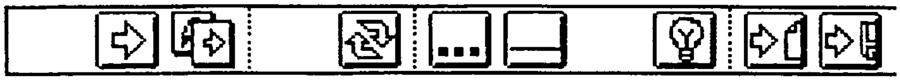
In edit mode, the editing tools become available on the *edit mode palette* below the window menu bar, as shown below. See the *Tools* section of this chapter for more information.



mode button
(in edit mode)

When you are ready to test the VI, click on the *mode button* or select **Change to Run Mode** from the **Operate** menu. This compiles your VI and puts the VI in run mode. At this point you can set debugging options, run the VI, place the VI in continuous run mode, or print data. If you just want to run the VI from edit mode without setting other options in the run mode palette, click on the run arrow. If necessary, LabVIEW compiles the VI first, then switches the VI to run mode, runs the VI, and switches back to edit mode after the VI has run.

The *run mode palette* is shown in the following illustration. In run mode, various debugging options become available. Refer to Chapter 3, *Executing VIs*, for more information on debugging options.



mode button
(in run mode)

If you want to switch to edit mode from run mode, click on the mode button or select **Change to Edit Mode** from the **Operate** menu. If you run the VI from run mode, the VI is still in run mode after executing.

Tools

A *tool* is a special operating mode of the mouse cursor. You use tools to perform specific editing functions.

The *Operating* tool, shown at left, operates front panel controls (and indicators in edit mode). This is the only tool available in run mode.

The *Positioning* tool, shown at left, selects, moves, and resizes objects.

The *Labeling* tool, shown at left, creates and edits text labels.

The *Wiring* tool, shown at left, wires objects together on the block diagram and assigns controls and indicators on the front panel to terminals on the VI connector.

The *Coloring* tool, shown at left, colors various objects and backgrounds.

You change tools by clicking on the tool icon you want in the edit mode palette, or by pressing the tab key, which selects the next tool in sequence. You can also press the space bar to toggle between the Operating tool and the Positioning tool when the Panel window is active, and between the Wiring tool and the Positioning tool when the Diagram window is active.

Help Window

The LabVIEW Help window offers help information for functions, constants, subVIs, and controls and indicators. To display the window, choose **Show Help Window** from the **Windows** menu. You can move the Help window.

When you move a tool onto a function, a subVI node, or a VI icon (including the icon of the VI you have open, at the top right of the VI window), the Help window shows the icon for the function or subVI with wires of the appropriate data type attached to each terminal. Input wires point to the left, and output wires point to the right. Terminal names appear beside each wire. If the VI has a description associated with it, this description is also displayed. VI descriptions are entered in the GetInfo dialog for the VI.

If you do not have to wire a function input, the default value LabVIEW appears 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. All subVI node inputs are optional, and the default value 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 move a tool over a universal constant, the Help window displays the constant's value.

When you move a tool over a control or indicator, the Help window displays the description for that particular control or indicator.

When you place a tool over a wire, the Help window displays the data type carried by that wire.

If you hold a tool on a VI icon in a VI palette for a few moments, the Help window will display help for that VI. The delay is updating the help window from a VI palette is to avoid slowing down LabVIEW when you do not need help in selecting which VI you want to use.

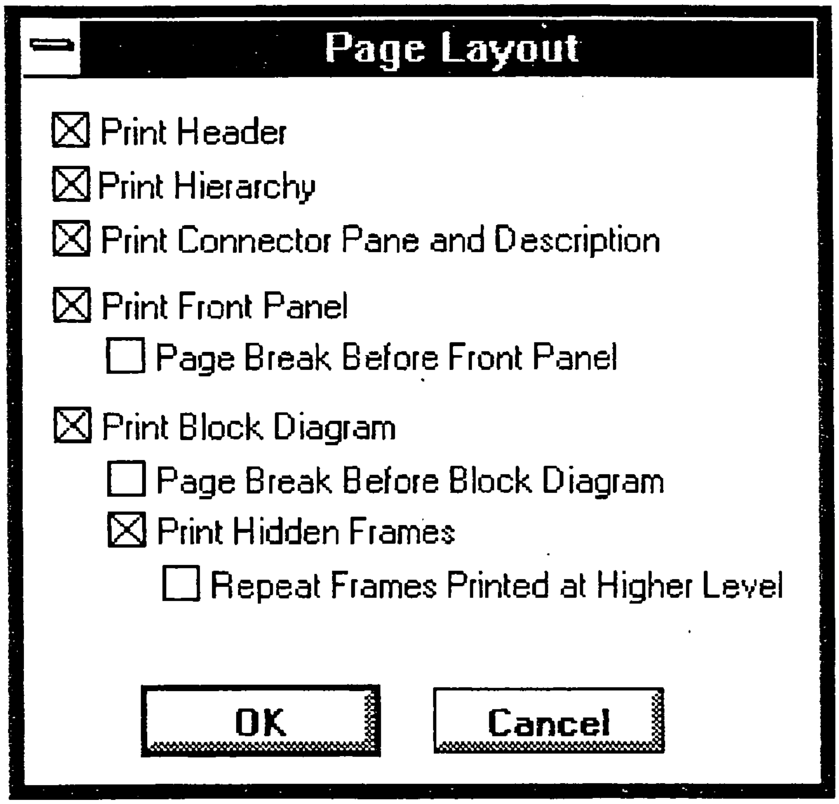
To close the Help window, click on its close box. See the *Online Help for Constants, Functions, and SubVI Modes* section of Chapter 11, *Introduction to the Block Diagram*, for more information on Help.

Printing VIs

You can print the front panel, block diagram, or other selected components of the current VI by choosing **Print...** from the **File** menu. You can also set up a VI to print its front panel at the end of execution (programmatic printing), as explained in the *Printing Programmatically* section in Chapter 3, *Executing VIs*.

To print a VI, make it the active window by clicking on its front panel or block diagram. Choose **Print...** from the **File** menu. Click on **OK** to print the VI according to the settings in **Page Layout**. Click on **Cancel** to return to the VI without printing.

To select printing options, choose **Page Layout...** from the **File** menu. The following dialog box appears.



Page Layout

☒ Print Header
☒ Print Hierarchy
☒ Print Connector Pane and Description
☒ Print Front Panel
    □ Page Break Before Front Panel
☒ Print Block Diagram
    □ Page Break Before Block Diagram
    ☒ Print Hidden Frames
        □ Repeat Frames Printed at Higher Level

The following list describes what happens if you mark the check boxes in the dialog box.

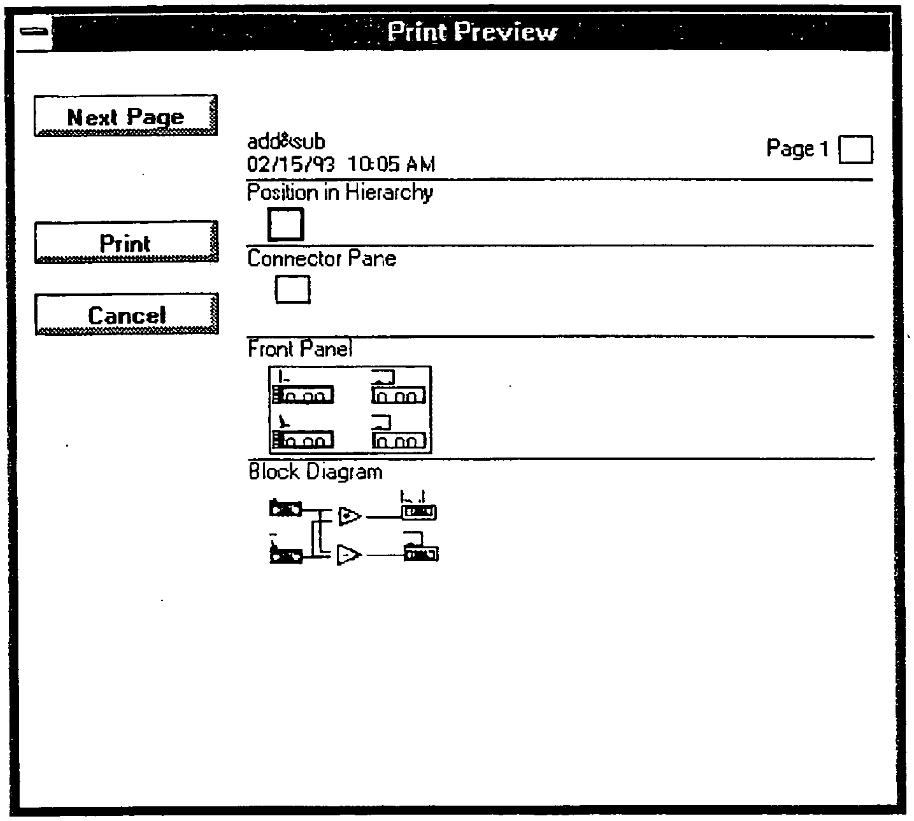
- **Print Header** prints the icon, VI name, page number, and date of the last save at the top of each page.

- **Print Hierarchy** prints a tree diagram of all VIs currently loaded into memory. This tree diagram is similar to the one that appears when you select **Show Hierarchy** from the **Windows** menu. VIs and their subVIs are connected by lines. The current VI is highlighted.

- **Print Connector Pane and Description** prints the VI icon/connector and the description entered in the **Description** dialog box.
- **Print Front Panel** prints the front panel of the VI.
  - **Page Break Before Front Panel** prints the front panel on a new page.
- **Print Block Diagram** prints the block diagram of the VI.
  - **Page Break Before Block Diagram** prints the block diagram on a new page.
  - **Print Hidden Frames** prints all the cases and frames of Case and Sequence structures in the block diagram of the VI. As LabVIEW prints each structure, its underlying frames are also printed, beginning with frame 0.
- **Repeat Frames Printed at Higher Level** may print subdiagrams of a structure more than once. For example, if a block diagram contains nested Sequence structures, LabVIEW prints the inner Sequence structure subdiagram currently displayed on the block diagram once when the entire block diagram is printed, once when the outer Sequence structure is printed (because the inner sequence structure is in one of the outer Sequence structure subdiagrams), and once when the inner Sequence Structure is printed. If this option is unchecked, each subdiagram prints only once. Subdiagram 0 of a structure always prints first, regardless of which subdiagram on the block diagram currently displays.

Previewing a VI

Select **Print Preview...** to display selected VI components as they will appear when printed, as shown in the following illustration.



Enhancing Printouts with Transparency and Decorations/Bitmaps

Controls in their standard configurations may not provide the kind of printout that you want in certain situations. For example, you may not want the shells of LabVIEW controls to be visible for reports, or you may want to include graphics, such as company logos.

In most cases, you can set up a subVI for printing in the desired format, as described in the section *Printing Programmatically* in Chapter 3, *Executing VIs*. By making the shells of controls transparent with the coloring tool, you can simplify the appearance of a front panel. The graphical objects from the **Decorations** menu are also useful for visually grouping objects. Finally, you can import bitmaps from paint programs to customize the panel for printouts.

Saving VIs

You save VIs through file dialog boxes that resemble dialog boxes in other applications on your operating system. If you save VIs as individual files, they must conform to the filenaming restrictions of your operating system. To avoid these restrictions and save disk space, save VIs in a compressed form in a file called a *VI library*. See the *VI Libraries* section for more information.

LabVIEW references VIs by name. You cannot have more than one VI or subVI with a given name in memory at one time. If you have two VIs with the same name, LabVIEW will load the first VI it finds without loading the other.

Save Options

You can save VIs with one of four save options in the **File** menu. Select the **Save** option to save a new VI, specifying a name for the VI and its destination in the disk hierarchy, or to save changes to an existing VI in a location previously specified.

Select the **Save As...** option to rename the VI in memory and to save a copy of the VI to disk. If you enter a new name for the VI, LabVIEW does not overwrite the disk version of the original VI. If you do not change the name of the VI in the dialog box, LabVIEW prompts you to verify that you want to overwrite the original file.

When you select the **Save A Copy As...** option, LabVIEW saves a copy of the VI in memory to disk under a different name, which you enter in the dialog box. 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 in memory to disk (the **Save Entire Hierarchy** option) and optionally save the VI, or VIs, without a block diagram to prevent further changes to the VI. You can save the VI, or VIs, without the block diagram only if you select **To New Location(s)**, which prompts you with new location options. This forces you to make a copy of the VI or VIs. If you select **To Current Location(s)**, LabVIEW does not prompt you to specify the location. To save a specified VI or VIs to a single location without being interrupted by multiple prompts, use **To VI Library**.

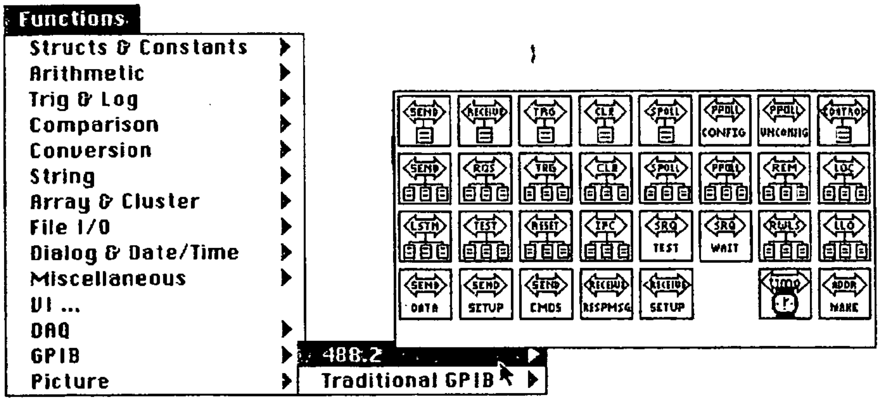
⚠ **Caution:** ***You cannot edit a VI after you save it without a block diagram. Always make a copy of the original VI.***

Revert

You can use the **Revert...** option in the **File** menu to return to the last saved version of the VI you are working on. A dialog box will appear to confirm whether to discard any changes in the VI.

VI Libraries

The VIs distributed with LabVIEW in the `vi.lib` directory are organized in VI libraries. These libraries appear as palettes in the **Functions** menu. For instance, the analysis VIs are contained in library files in a directory titled ANALYSIS. When you select **GPIB** from the **Functions** menu, a submenu with two options appears. When you select an option from that submenu, a palette of VIs appears.



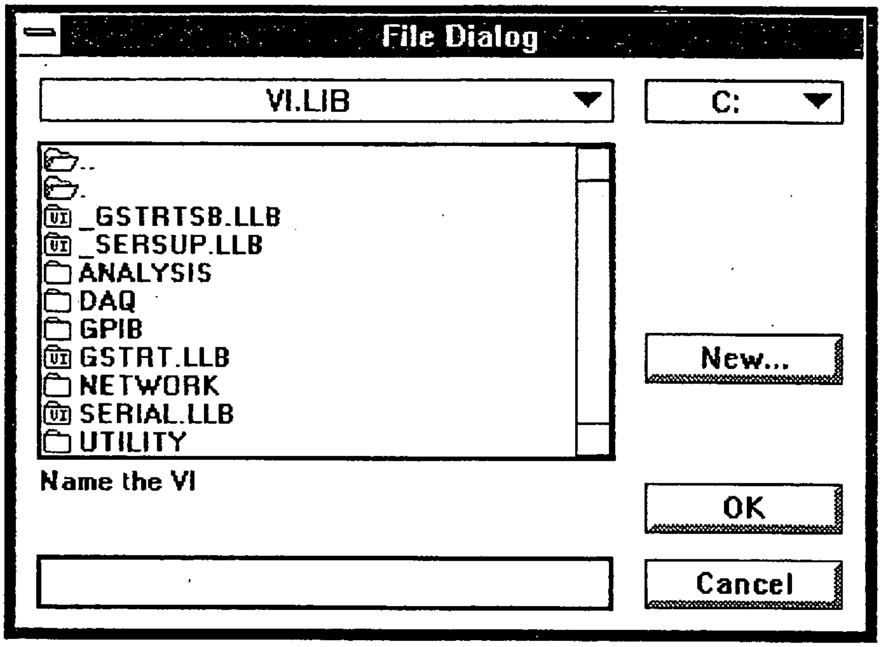
Note: ***Do not save your own VIs in the*** `vi.lib` ***directory. This directory is updated by National Instruments as needed during new version releases of LabVIEW. Placing VIs in*** `vi.lib` ***risks creating a conflict during future installations. Place VIs in*** `.lib` ***directories adjacent to, but not inside, the*** `vi.lib` ***directory instead.***

In file dialog boxes, a VI library appears as a folder with an extension of `.llb` appended to the name, as shown in the following illustration.

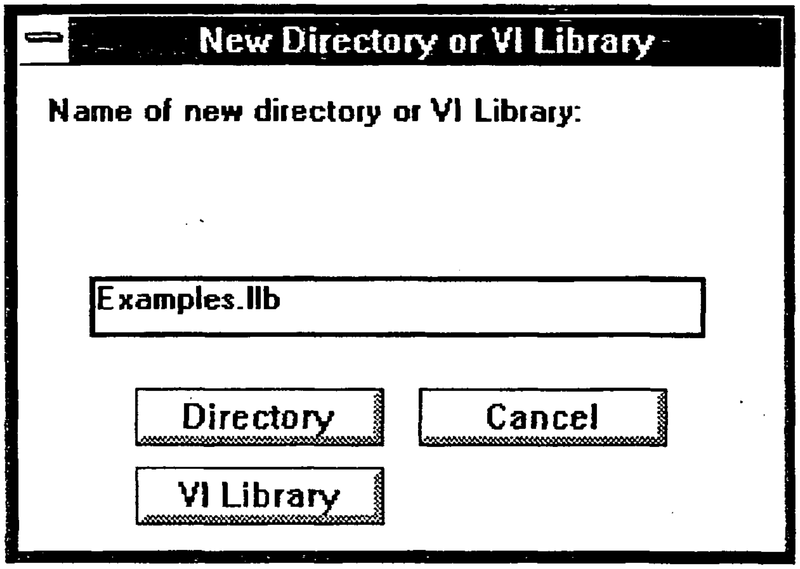
GSTRT.LLB
MYVI.LLB

Libraries have the same load, save, and open capabilities as directories. However, VI libraries are not directories; they are files. They cannot contain other VI libraries or directories. Libraries contain compressed versions of VIs. While compression and decompression time during loads and saves varies depending on drive speed and computer speed, the extra time libraries use is minimal, and the disk space they save is always substantial. Another advantage for some platforms is that VIs saved in a library are not bound by the operating system naming restrictions. VI names can contain up to 31 characters, including spaces and the `.vi` extension, and are not case sensitive. Also, libraries are more portable among different platforms. The library name must conform to the operating system filename restrictions.

You create a VI library by clicking on the **New...** button of the **Save**, **Save as...**, or **Save a Copy As...** dialog box, as shown in the following illustration.



Enter the name of the new library in the dialog box that appears and append an .llb extension. Then click on the **VI Library** button. If you do not include the .llb extension, LabVIEW adds it.



Name of new directory or VI Library:

Examples.llb

You can remove a file from a VI library using buttons in the dialog box displayed by the **Edit VI Library...** option of the File menu.

Accessing Your VI Library from the Functions Menu

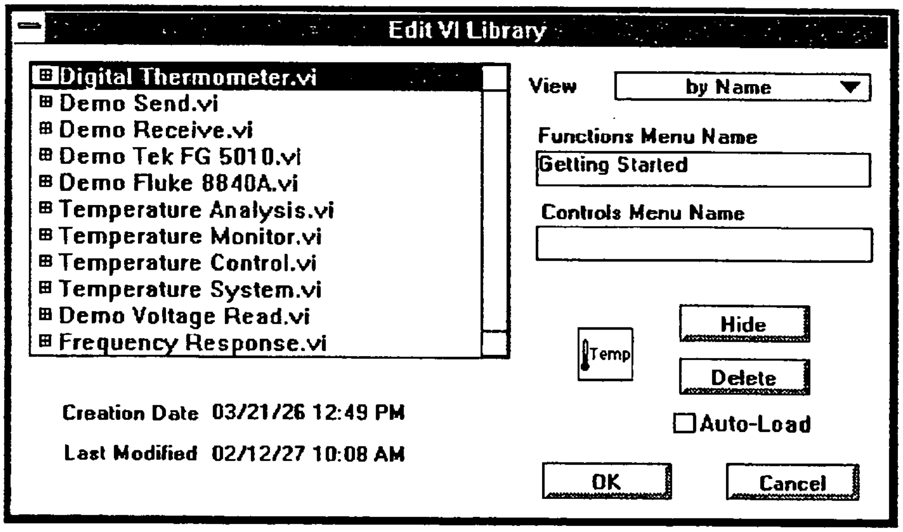
When you start LabVIEW, the application looks in the LabVIEW directory for directories ending in .lib. LabVIEW builds palette menus for all VI libraries or directories inside of .lib directories. If you want your VIs to appear in the **Functions** menu, create your own directory ending in .lib, and place directories or VI libraries in that directory. When you restart LabVIEW, it creates palettes for each of these directories and VI libraries and adds them to the **Functions** menu. LabVIEW arranges VIs alphabetically for directories. For VI libraries, LabVIEW arranges VIs according to the layout you create with the **Edit VI Library...** option in the **File** menu.

Your VIs must be in a directory or a VI library within the .lib directory. LabVIEW will not display an individual VI on a palette if it is saved directly into the .lib directory.

Edit VI Library Dialog

You can use the **Edit VI Library** dialog box to edit the contents of a VI library. You can selectively delete files from a VI library. You can also define control and function palettes, and palette names, to use for menus if the VI library is placed in a `.lib` library directory (see the *Saving a Custom Control* and *Adding a Custom Controls Menu* sections of Chapter 4, *Introduction to Front Panel Objects*, for more information). When you define the palettes, you do not have to have all VIs and controls visible in the palette; you can hide or show the VIs and controls you want.

The Edit VI Library dialog box initially displays a list of the files in the VI library. As you move through the list, the creation date and last modification date for the selected file is shown at the bottom of the dialog box.


Edit VI Library

⊞ Digital Thermometer.vi
⊞ Demo Send.vi
⊞ Demo Receive.vi
⊞ Demo Tek FG 5010.vi
⊞ Demo Fluke 8840A.vi
⊞ Temperature Analysis.vi
⊞ Temperature Monitor.vi
⊞ Temperature Control.vi
⊞ Temperature System.vi
⊞ Demo Voltage Read.vi
⊞ Frequency Response.vi

View: by Name ▼

Functions Menu Name: Getting Started

Controls Menu Name:

Creation Date 03/21/26 12:49 PM

Last Modified 02/12/27 10:08 AM

□ Auto-Load

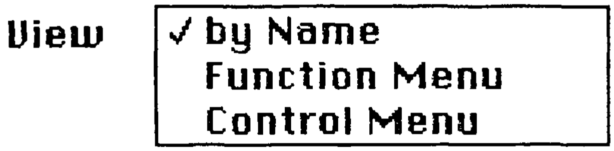
⊞
file visible

Icons at the left side of the list indicate the status of each file. The icon shown at the left indicates that the file is visible in a palette. You can add a file to the palettes or remove it by using the Hide/Show button.

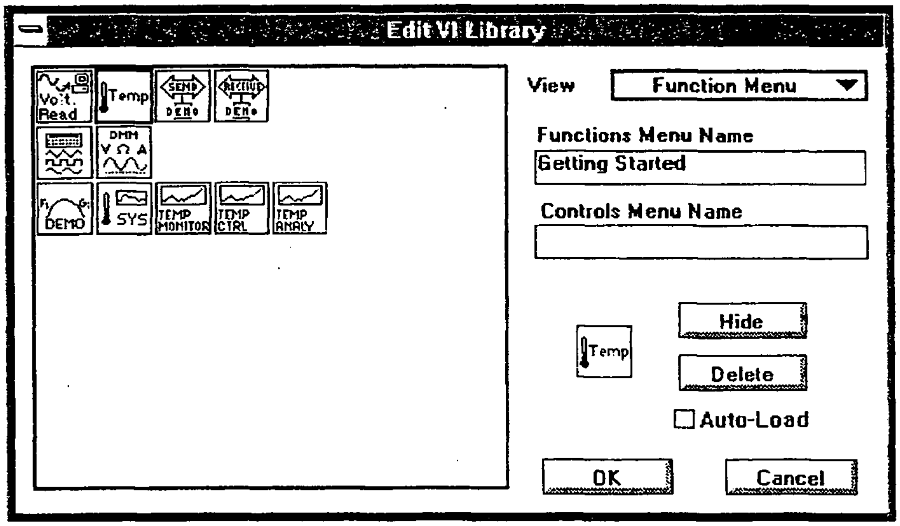
file marked for deletion

The icon shown at the left indicates that the file is marked for deletion. You select and deselect files for deletion with the Delete/Restore buttons. Marked files are only deleted if you select the OK button.

You can view the function palette menu (used in the **Functions** menu for placing VIs and globals) or the control palette menu (used in the **Controls** menu for placing custom controls and typedefs) by using the **View** popup menu.



If you select one of the palette menus, the list is replaced by an icon grid showing the arrangement of the icons you've selected for visibility in the palette menu. The menu name used for the palette is determined by the **Functions Menu Name** and **Controls Menu Name** fields.



View: Function Menu

Functions Menu Name: Getting Started

Controls Menu Name:

□ Auto-Load

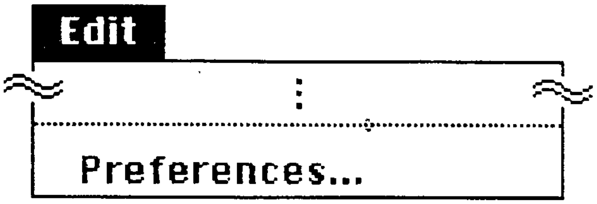
The **Auto-Load** option has two uses. If you are creating an application, selecting **Auto-Load** controls which VIs open automatically when you run the application.

If you launch LabVIEW with a command line option specifying a particular library, selecting Auto-Load determines which VIs load automatically.

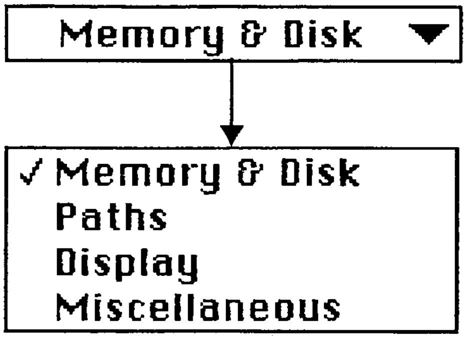
Configuring LabVIEW

The preferences dialog lets you configure a number of default parameters and customize LabVIEW. These options let you set up the application memory, specify LabVIEW search paths, and change default options.

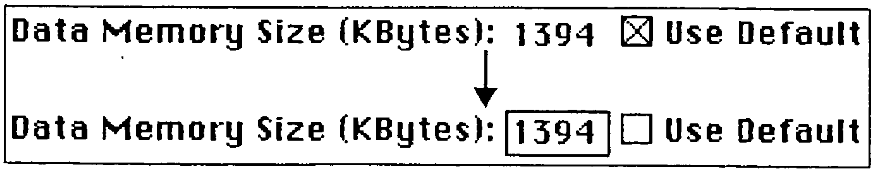
You access the preferences dialog by selecting **Preferences...** from the **Edit** menu.



The dialog initially displays a set of preferences that let you configure memory and disk space requirements. The pull-down menu at the top of the dialog lets you select among the different categories of preferences.



Most of these options have a default value, which is initially displayed. To change the default, deselect the **Use Default** check box and specify the new value.



Data Memory Size (KBytes): 1394 ☒ Use Default

Data Memory Size (KBytes): 1394 □ Use Default

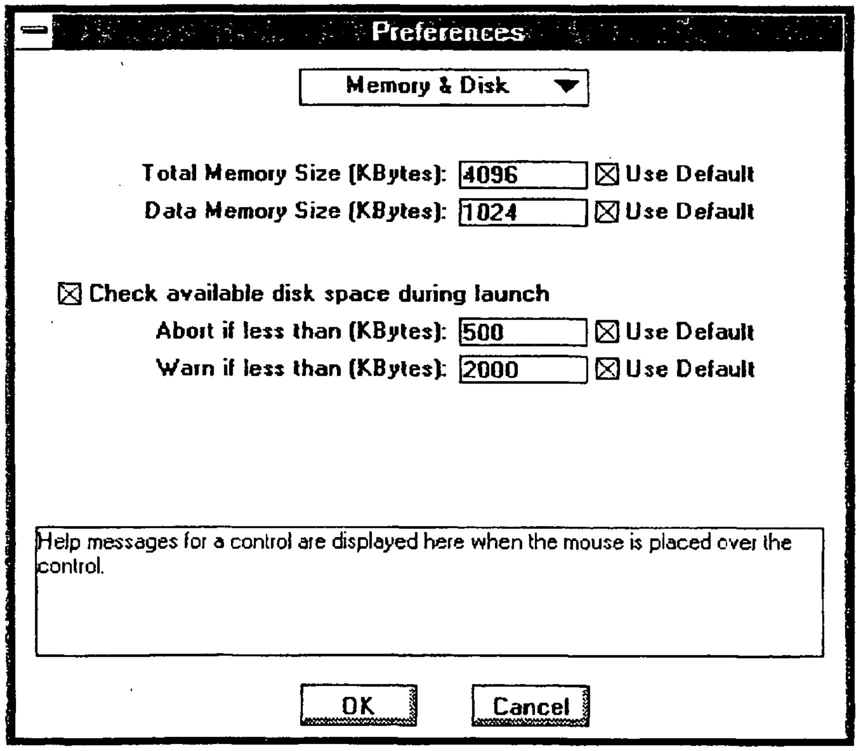
How Preferences are Stored

You will not usually have to edit preference information manually, or know its exact format, because the preferences dialog takes care of it for you.

Preference information is stored in a `LabVIEW.INI` file in your WINDOWS 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 will check to see if there is a `[LabVIEW]` section of your `WIN.INI` file, and if so, check for the configuration value in that file.

Memory and Disk Preferences

You configure memory and disk space requirements using the following page of the preferences dialog.



Total Memory Size (KBytes): 4096 ☒ Use Default
Data Memory Size (KBytes): 1024 ☒ Use Default

☒ Check available disk space during launch
Abort if less than (KBytes): 500 ☒ Use Default
Warn if less than (KBytes): 2000 ☒ Use Default

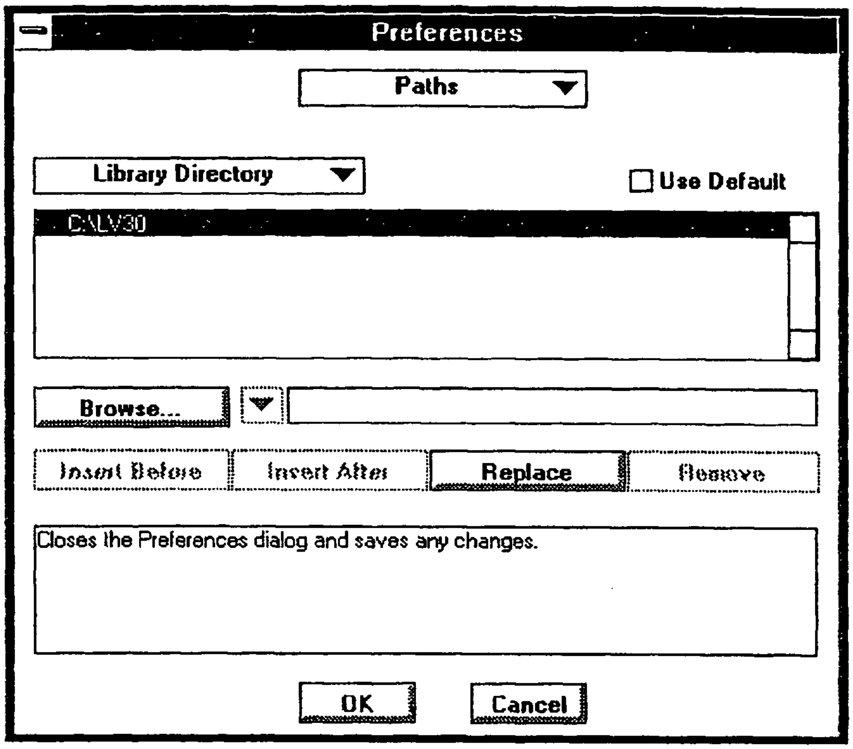
Help messages for a control are displayed here when the mouse is placed over the control.

When you launch LabVIEW, it will allocate a block of memory of the size selected for **Total Memory Size.** LabVIEW uses this memory for all of its memory allocation. Of this number, some is reserved strictly for use by the diagram for storing data. The rest is used by the application. **Data Memory Size** specifies how much memory out of the total available is allocated for storing data.

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 disk space at launch time. If you have less than 500 KB of disk space free, LabVIEW will alert you and quit. If you have less than 2 MB available, LabVIEW will warn you, but continue. You can turn these warnings off or change the levels at which they alert you using the options at the bottom of this dialog.

Path Preferences

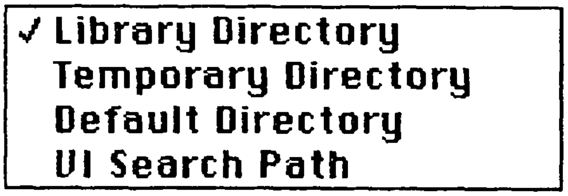
You can specify the directories that LabVIEW searches when searching for VIs as well as the paths LabVIEW uses for temporary files and the library directory. Use the **Paths** page of the **Preferences** dialog.



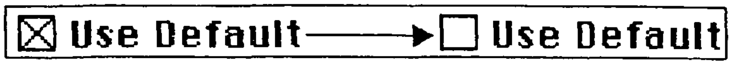
C:\LV30

Closes the Preferences dialog and saves any changes.

At the top of this dialog box is a pull-down menu you can use to select the path you want to view or edit.

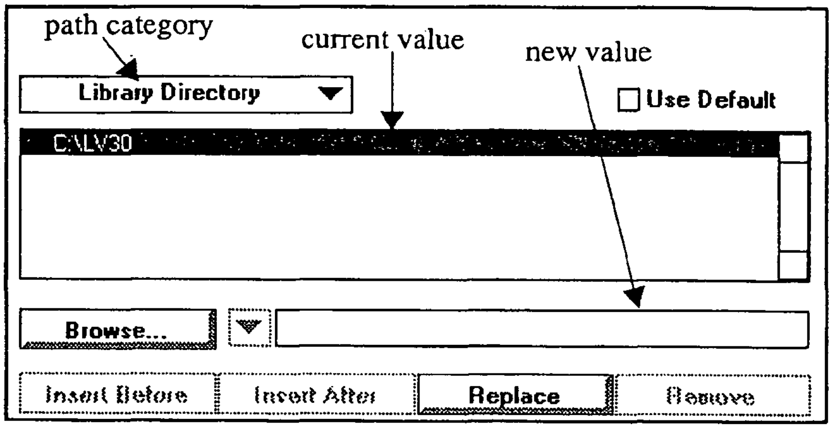


Initially, the controls in the middle of this dialog that let you edit a path are grayed out. If you want to change one of these preferences, deselect the **Use Default** check box.



Library, Temporary, and Default Directories

The **Library Directory**, **Temporary Directory**, and **Default Directory** are single directories. When you edit one of these paths, you are given options that let you type in a new path and replace the existing path, or browse using the file dialog box to select a path.



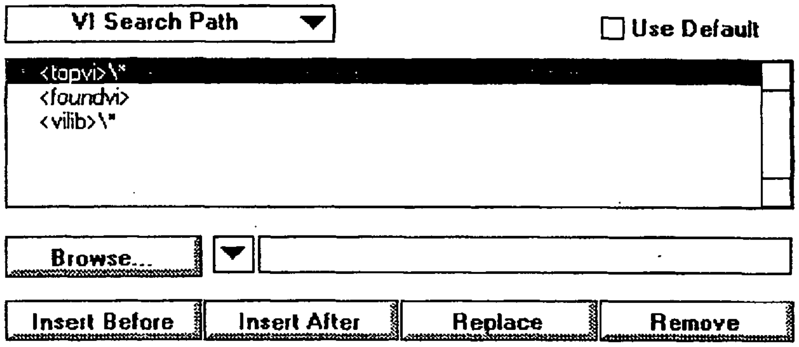
The **library directory** specifies the absolute pathname to the directory containing vi.lib and any library directories you supply. The default is the directory containing LabVIEW.

The **temporary directory** specifies the absolute pathname to the directory for temporary files. The default is the directory containing LabVIEW.

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 also returns this value. The default is the current working directory.

VI Search Path

The VI Search Path is the list of directories that LabVIEW searches when looking for a VI, control, or external subroutine. 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.



VI Search Path ▼

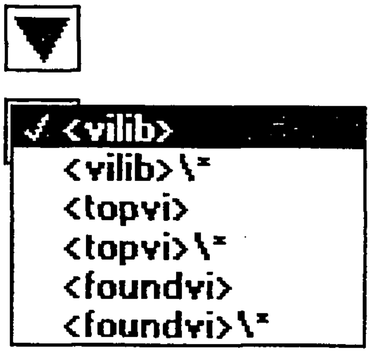
□ Use Default

<topvi>*
<foundvi>
<vilib>*

The list indicates paths that LabVIEW will search, 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 the directory's subdirectories. You can make the search hierarchical by appending a `*` as a new path item. For example, to search the directory `C:\VIs\` recursively, you would enter `C:\VIs\*` as the path.

The special pull-down menu, shown in the following illustration, lets you select from several special directories.

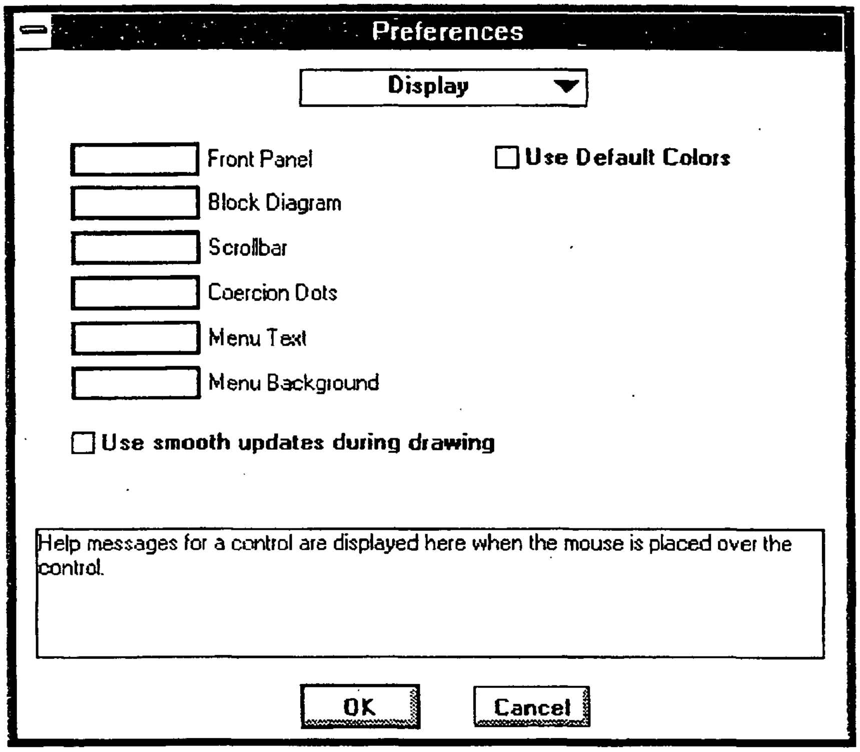


`<vilib>` as a path entry is a symbolic path that refers to the `vi.lib` directory inside of the Library Directory. `<topvi>` is a symbolic path that refers to the directory containing the top level VI. `<foundvi>` refers to a list of directories that LabVIEW creates when you open a VI. If, during a search, you manually select a directory, that directory is added to the list of `foundvi` directories. 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.

Display Preferences

You configure default colors from the display preferences dialog. You can also configure LabVIEW to use smooth updates on all controls.



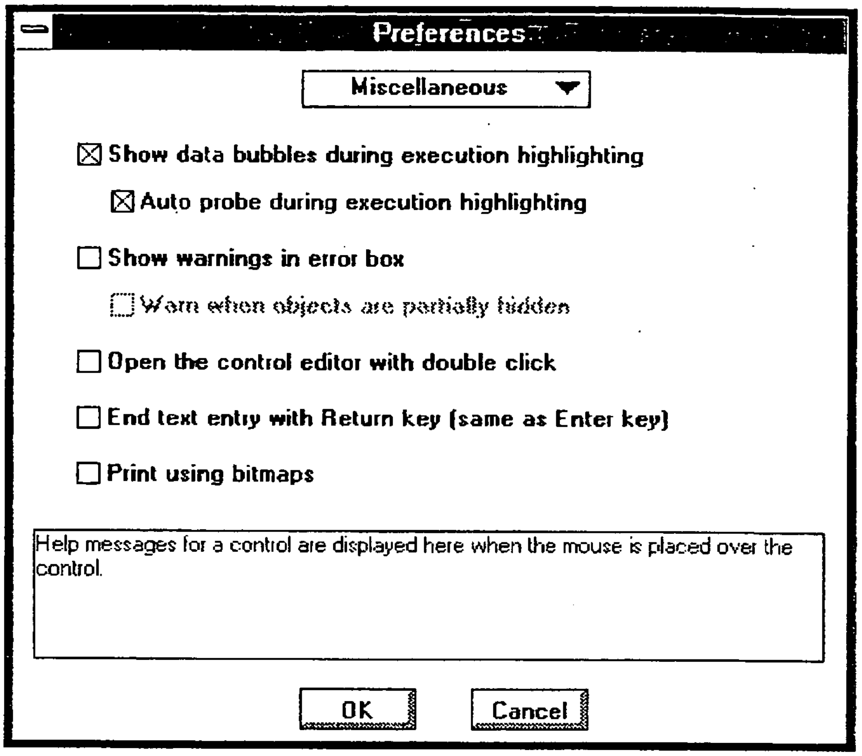
Help messages for a control are displayed here when the mouse is placed over the control.

The **Front Panel** and **Block Diagram** colors are the default colors for new windows. Thus, changing these parameters does not affect old VIs. The **Scrollbar** color, **Menu Bar** colors, and **Coercion Dots** colors affect all VIs that are currently open.

The **Use smooth updates during drawing** option lets you configure LabVIEW to use offscreen drawing to smooth out displays. When LabVIEW updates a control with smooth updates off, it erases the controls contents 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.

Miscellaneous Preferences

The miscellaneous page of the preferences dialog lets you configure options for execution highlighting, configure warning levels for the error box, edit a control using a double click, end text entry with an Enter key, and turn Bitmap printing on by default.



☒ Show data bubbles during execution highlighting

☒ Auto probe during execution highlighting

□ Show warnings in error box

□ Warn when objects are partially hidden

□ Open the control editor with double click

□ End text entry with Return key (same as Enter key)

□ Print using bitmaps

Help messages for a control are displayed here when the mouse is placed over the control.

The default selection Show **data bubbles during execution highlighting** animates execution flow by drawing bubbles along the wires. You may want to turn off this feature if it slows down performance too much on your computer.

The default selection **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.

The selection **Show warnings in error box** optionally displays warnings in addition to errors. A warning does not mean that the VI is incorrect; it just helps by pointing out a potential problem in your diagram. One of the warnings alerts you that there are overlapping objects. You may want to restrict this warning to warn only about objects that are completely hidden.

You normally access the control editor by selecting a control and then selecting **Edit Control** from the **Edit** menu. You can select **Open the control editor with double click** so that double-clicking on a control will open the control editor. By default, this option is turned off, because the control editor is generally used only by advanced users.

When you enter data, the Enter key on the alphanumeric keyboard embeds a newline in string controls. The Enter key on the numeric keypad ends text entry. You can select **End text entry with Return key (same as Enter key)** so that both keys function like the Enter key on the numeric keypad, and end text entry. If you configure LabVIEW in this manner, you can embed newlines by hitting <Ctrl-Enter>.

LabVIEW supports two modes of printing—bitmap printing and nonbitmap printing. Nonbitmap printing generally produces a higher resolution printout. It usually requires less memory and produces printouts faster than bitmap printing. However, due to font substitution and the way that some printer drivers handle overlapping text and graphics, it may not produce an image that looks exactly like what is on the screen. Bitmap printing may produce a more accurate printout by creating a bitmap image of the screen and sending it to the printer. Notice that this can be more costly in terms of time and memory. Nonbitmap printing is normally the default setting. You can optionally make bitmap printing the default using the **Print using bitmaps** option.

Chapter 2

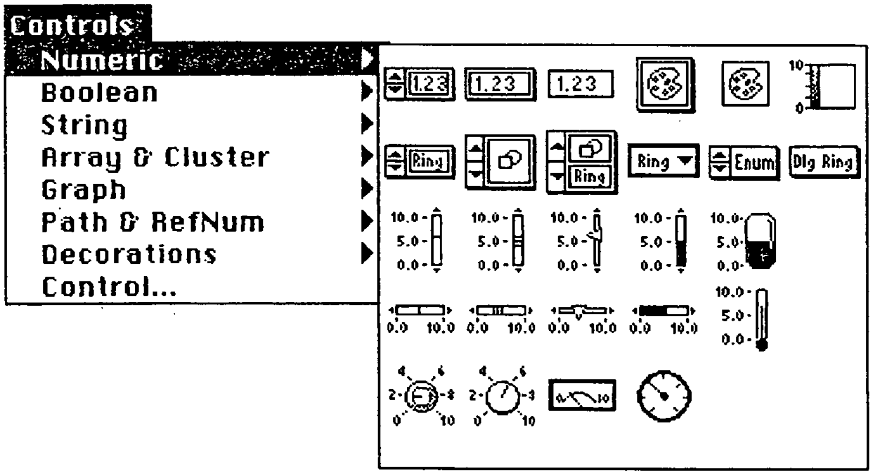
Editing VIs

This chapter discusses editing techniques for the front panel and the block diagram.

When you are familiar with these techniques, see the Block Diagram Programming part of this manual (Chapters 11 through 18) for help in building block diagrams, and see the Front Panel Objects part (Chapters 4 through 10) for information on building front panels. This chapter concludes with a section explaining how to edit the icon and connector.

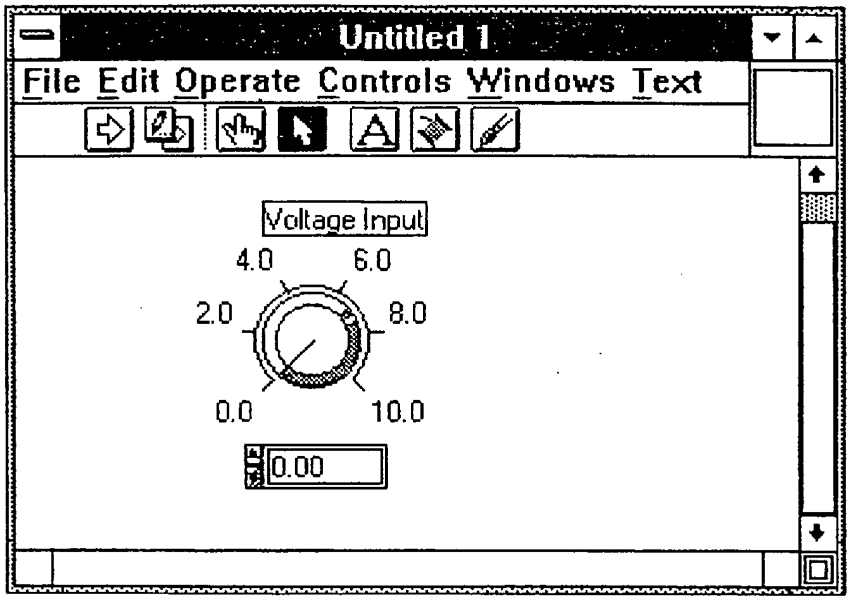
Creating Objects

Starting with an untitled Panel window, select an object from the **Controls** menu.

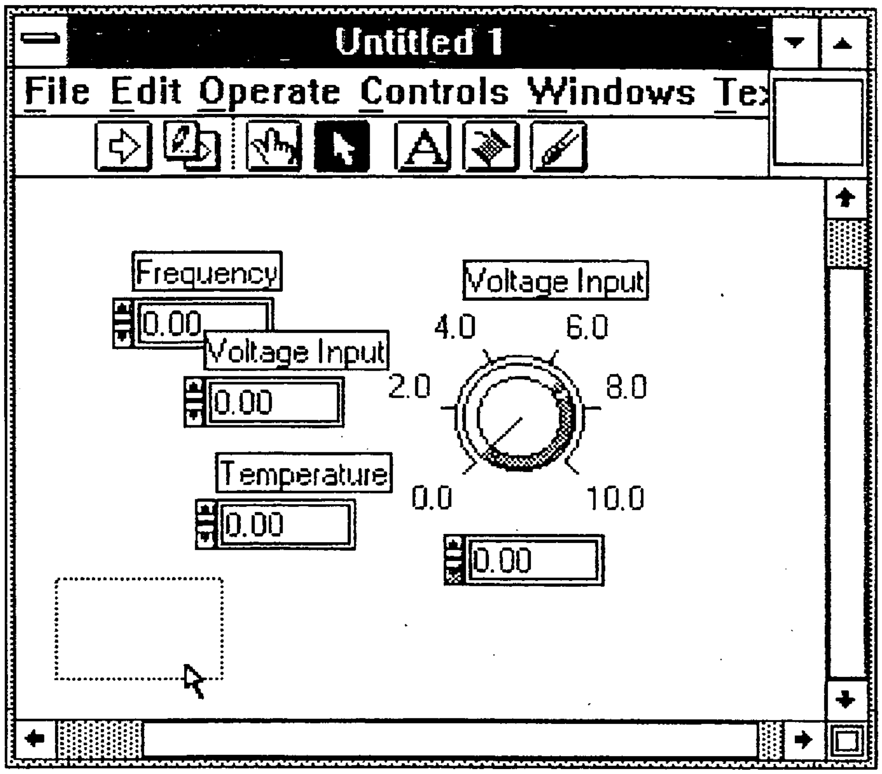


The object appears in the panel window with a black or gray rectangle representing a label. If you want to retain the label at this time, enter text from the keyboard. After you enter text into a label, any of the following actions completes the entry:

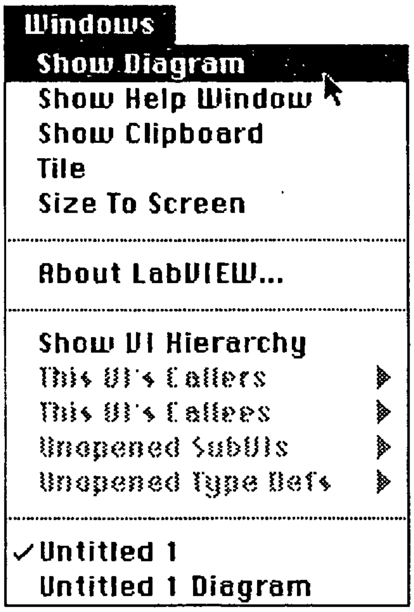
- Press <Shift-Enter>.
- Press <Enter> on the numeric keypad.
- Click on the enter button in the tool palette.
- Click outside the label.

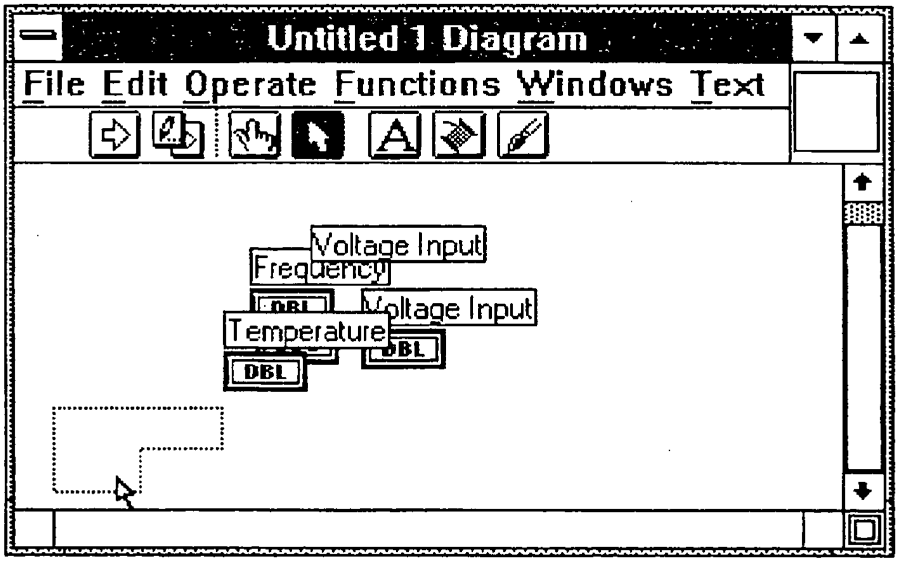


For the purposes of this discussion, place two more objects on the front panel from the **Controls** menu. To move an overlapping object, click on the object (not its label) with the Positioning tool and drag the object to a new location.



Select **Show Diagram** from the **Windows** menu to see the terminals of the components you just placed in the Panel window. Move terminals so that they do not overlap.

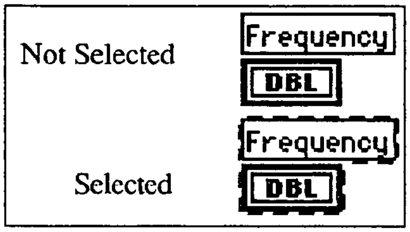




Selecting Objects

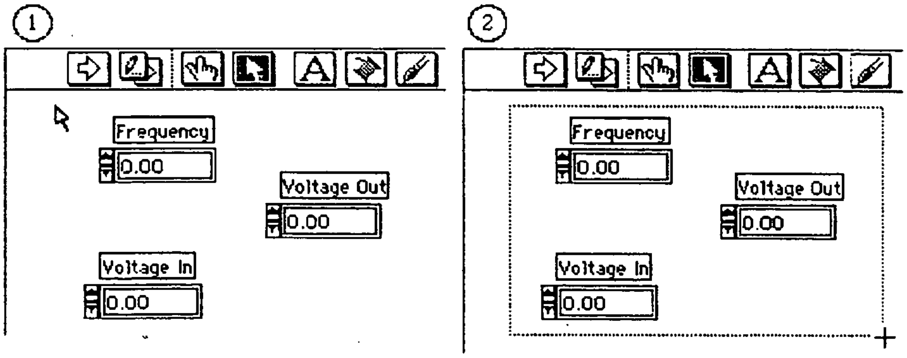
You must *select* an item before you can manipulate it.

To select an object, click the mouse button while the Positioning tool 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.



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.

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.

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 small, precise increments by pressing the appropriate arrow key on the keyboard once for each pixel you want the objects to move. Hold down the arrow keys to repeat the action.

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 line 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.

Duplicating Objects

To duplicate a LabVIEW object, select the object and choose **Copy** from the **Edit** menu. Then click where you want to place the duplicate and choose **Paste**. You can duplicate several objects at the same time by dragging a selection marquee around the items before duplicating them.

Another method of duplicating an object is *cloning*, which is described in detail in your tutorial manual.

You can duplicate most objects in LabVIEW. However, you cannot duplicate front panel control and indicator terminals in the diagram except by copying the associated front panel control or indicator, nor can you copy parts that are attached to a particular object.

Copying Objects between VIs or from Other Applications

You can copy and paste objects from one VI to another or copy or cut pictures or text from other applications to LabVIEW by using the **Copy**, **Cut**, and **Paste** commands from the **Edit** menu. These operations place the copy on the Clipboard. When an object is on the Clipboard, you can paste it into another part of the current VI, another VI, or another application. Pictures pasted into LabVIEW from other applications have their own pop-up menus just like other LabVIEW objects.

Deleting Objects

To delete an object, select the object and choose **Clear** from the **Edit** menu or press the backspace or delete key. Block diagram terminals for front panel controls and indicators disappear only when you delete the corresponding front panel controls or indicators.

Although you can delete most objects, you cannot delete control or indicator components such as labels and digital displays. However, you can hide these components by selecting **Show** from the pop-up menu and then deselecting the **Label** or **Digital Display** options.

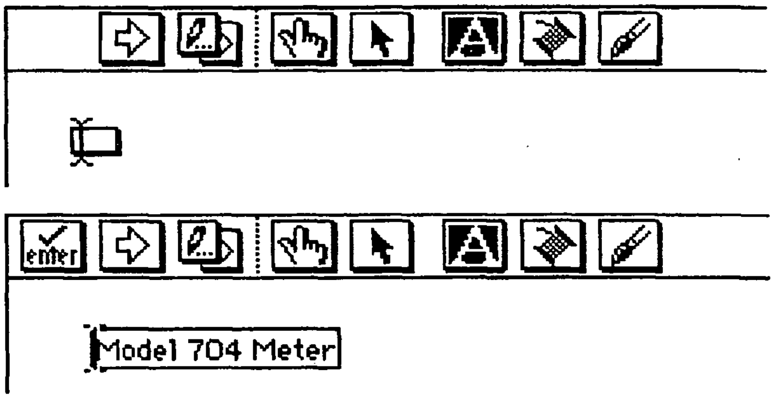
Labeling Objects

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 panels and 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.

Labeling tool

Creating Labels

To create a free label, select the Labeling tool from the edit mode palette and click anywhere in empty space.



enter button

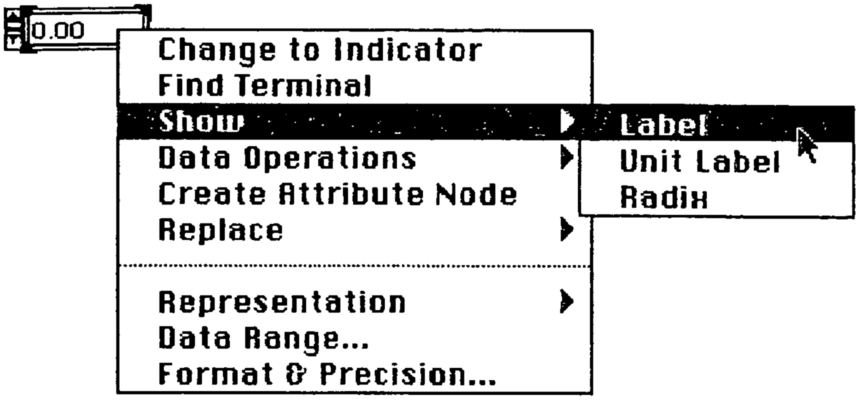
A small, bordered box appears with a text cursor at the left margin ready to accept typed input. Type the text you want to appear in the label. The *enter* button appears on the tool palette to remind you to end text entry by clicking on the enter button, pressing <shift-enter>, pressing <enter> on the numeric keypad, or clicking outside the label. If you do not type any text in the label, the label disappears as soon as you click somewhere else.

If you place a label on a control or indicator with moving parts, such as a meter, it will slow down screen updates and make the control or indicator flicker. To avoid this problem, do not overlap front panel objects with a label.

You can also copy the text of a label by selecting it with the Labeling tool. Double-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 **Copy** from the **Edit** menu to paste a copy of the text onto the Clipboard. Now you can highlight the text of a second label and select **Paste** from the **Edit** menu to replace the 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, and select **Paste** from the **Edit** menu.

When you create a control or indicator on the front panel, a blank, owned label accompanies it, awaiting input. The label disappears if you do not enter text into it before clicking elsewhere with the mouse.

To display a hidden label, pop-up on the object and select **Label** from the **Show** submenu of the pop-up menu, as shown below.

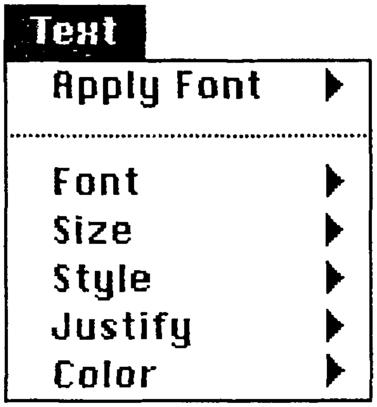


The bordered label appears, waiting for typed input. The label disappears if you do not enter text into it before clicking outside the label.

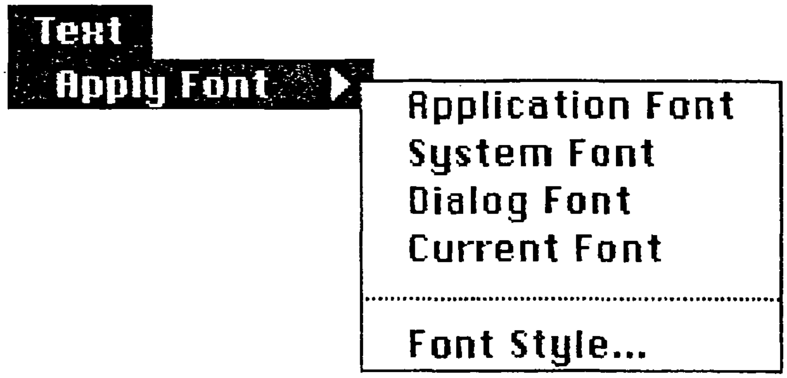
Note: ***SubVIs do not have labels. They have names, which look just like labels but which cannot be edited. Function labels, on the other hand, are empty so that you can edit them to reflect the use of the function in the block diagram. For example, you can use the label of an Add function to document what quantities are being added, or why they are being added at that point in the block diagram. See the note in the*** *Creating Descriptions* ***section in this chapter for related information.***

Changing Font, Style, Size, and Color of Text

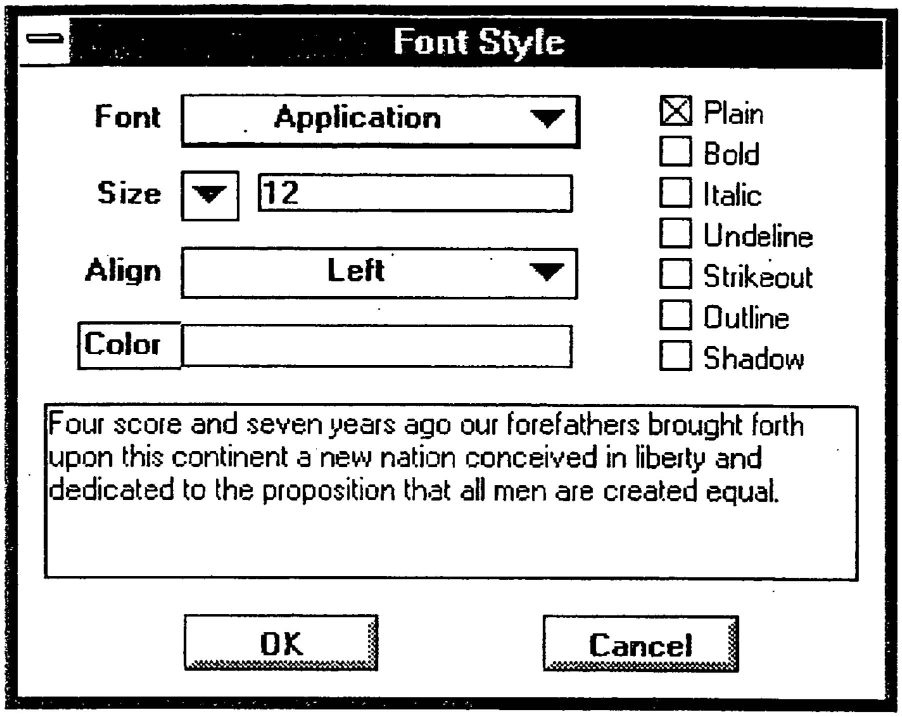
You can change text attributes in LabVIEW using the options on the **Text** menu. If you select objects or text and make a selection from this menu, the changes apply to everything selected. If nothing is selected, the changes apply to the default font.



The **Apply Font** menu lets you choose from one of three predefined font styles, as well as the last font used. There are three fonts: Application, System, and Dialog. Each of these font options has an associated menu hotkey, so you can change fonts quickly.



The **Apply Font** menu also has a **Font Style...** option which displays a dialog box for font selection. The dialog box is quicker than the menu if you need to make several changes at once.



Font Style

Font: Application

Size: 12

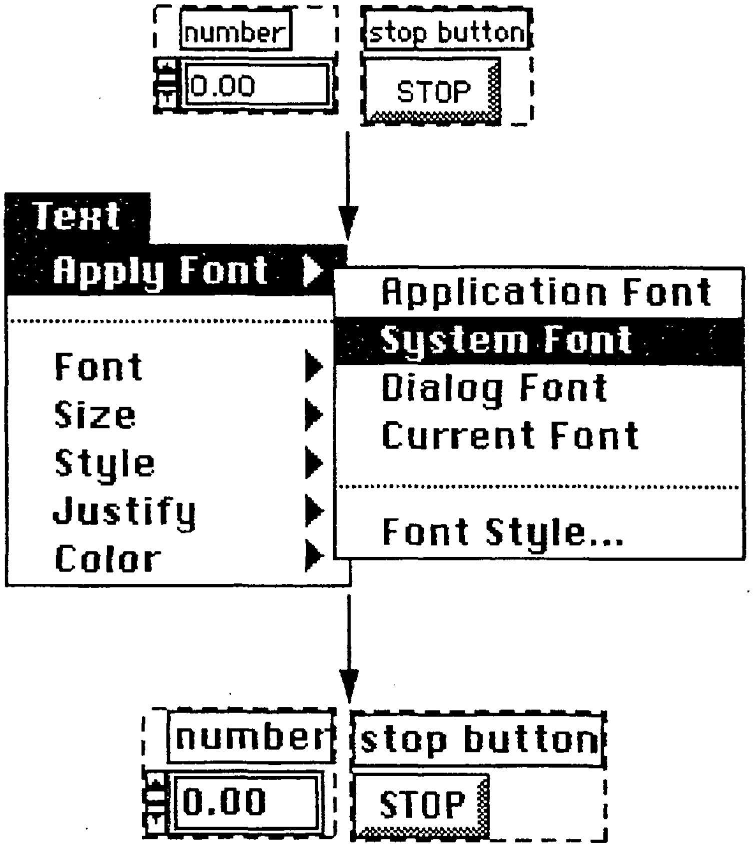
Align: Left

Color

☒ Plain
☐ Bold
☐ Italic
☐ Undeline
☐ Strikeout
☐ Outline
☐ Shadow

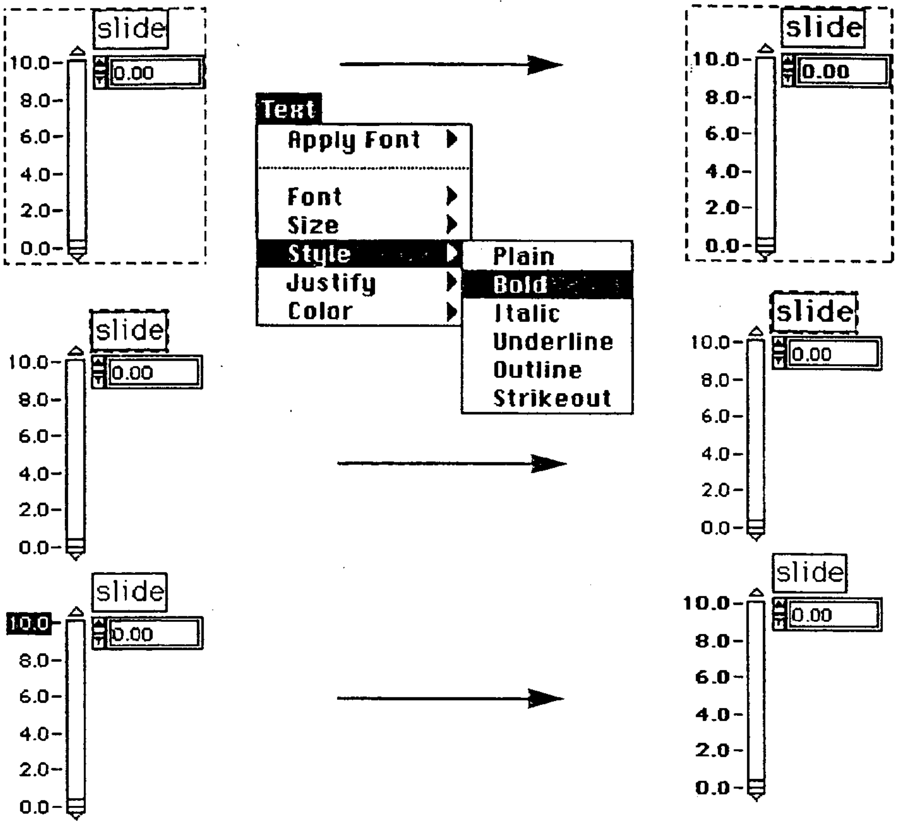
Four score and seven years ago our forefathers brought forth upon this continent a new nation conceived in liberty and dedicated to the proposition that all men are created equal.

Any font selections made from the **Text** menu apply to all of the objects in the current selection. 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 to the System font.



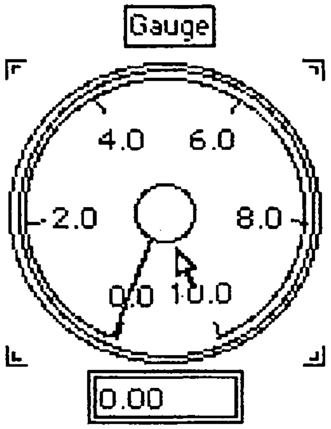
Notice that as many font attributes as possible are preserved when you make a change. If you change several objects to Courier font, they will retain their size and styles if possible. In the same way, changing the size of multiple text selections does not change the selections to the same font. These rules do not apply if you make your changes with the **Apply Font** menu, which changes the selected objects to have 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. If you select the slide and select **Bold**, then the scale, digital display, and label all change to a bold font. If you select only the label, then only the label changes to bold. If you select text from the scale, then scale markers, which must all have the same font, will change to bold as shown in the following illustrations.



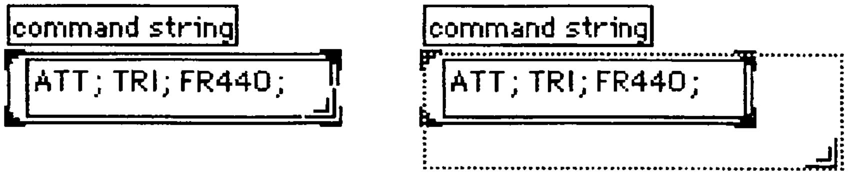
Resizing Objects

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 below.



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.


command string

ATT; TRI; FR440;

command string

ATT; TRI; FR440;

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.

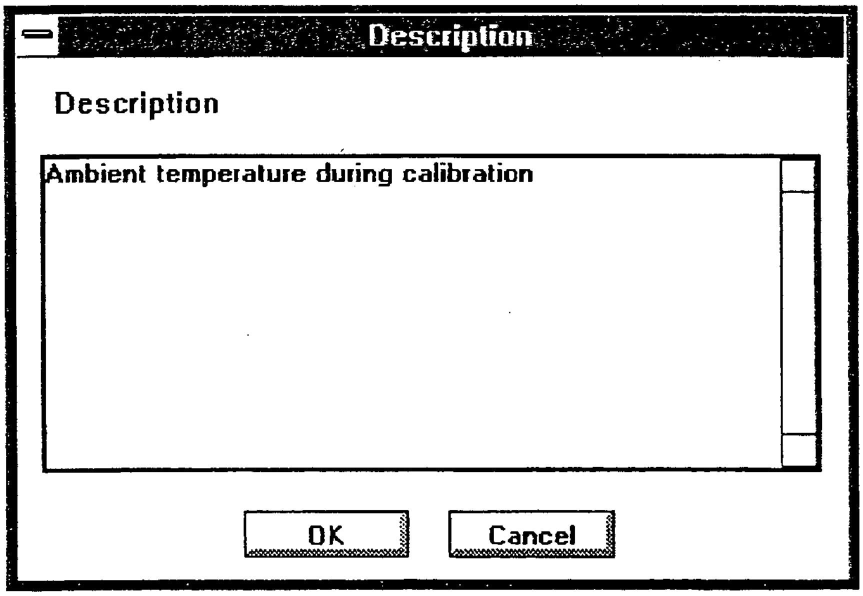
Some objects can grow horizontally or vertically only. The Resizing cursor appears the same but the dotted grow outline will move in only one direction. To restrict growth vertically or horizontally, or to maintain the current proportions, hold down the shift key as you click and drag.

Resizing Labels

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.

Creating Descriptions

If you want to enter a description of a LabVIEW object, such as a control or indicator, choose **Description...** from the **Data Operations** submenu of the object's pop-up menu. Enter the description in the dialog box shown below and click **OK** to save it. LabVIEW displays this description whenever you subsequently choose **Description...** from the object's pop-up menu.



Ambient temperature during calibration

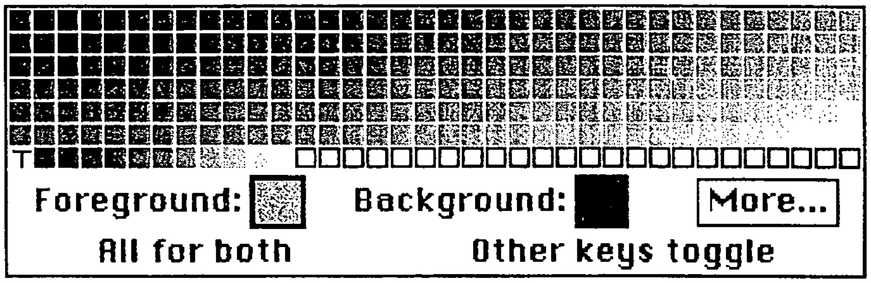
Note: ***You cannot edit subVI descriptions from the calling VI diagram. You can edit a VI description through the Get Info... selection of the File menu when the VI's front panel is open. Function description boxes are blank so that you can describe the action taking place at each occurrence of that 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.

Coloring tool

To change the color of an object or the background screen, pop-up on it with the Coloring tool, as shown at the left. The following palette appears in color.



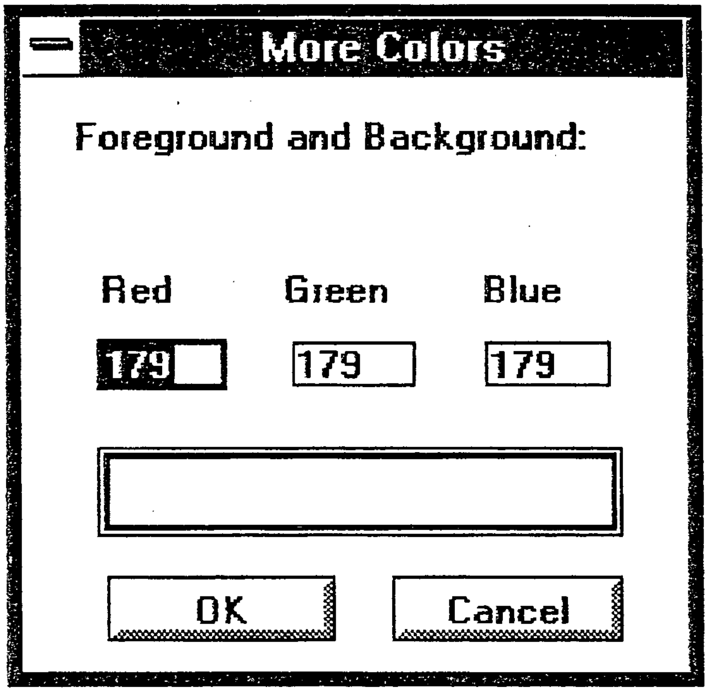
As you move through the palette while pressing the mouse button, the object or background you are coloring redraws with the color the cursor is currently touching. This gives you a preview of the object in the new color. If you release the mouse button on a color, the object retains the selected color. To cancel the coloring operation, move the cursor out of the palette before releasing the mouse button.

If you select the box with a *T* in it, LabVIEW makes the object transparent. With this feature, you can layer objects. For instance, you can place invisible controls on top of indicators; or you can create numeric controls without the standard three-dimensional container. Transparency affects only the appearance of an object. The object responds to mouse and key operations as usual.

Some objects 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 coloring tool.

Selecting the **More** option from the coloring palette calls up a dialog box with which you can customize the colors.



More Colors

Foreground and Background:

Red: 179
Green: 179
Blue: 179

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 color's display and enter the new value, or click on the appropriate arrow button 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 Coloring tool sets that object to the current color.

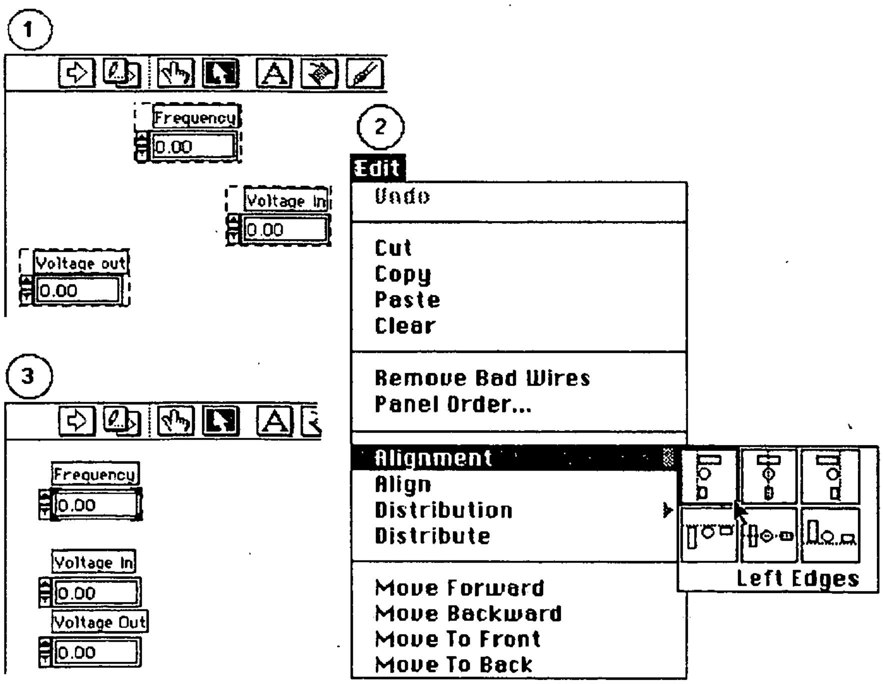
You can also duplicate the color of one object and transfer it to a second object without going through the color palette. <Ctrl>-click with the Coloring tool on the object whose color you want to duplicate. The Coloring tool appears as an eye dropper and takes on the color of the selected object. Now you can click on another object with the Coloring 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**, **Distribution**, **Align**, and **Distribute** items in the **Edit** menu.

Aligning Objects

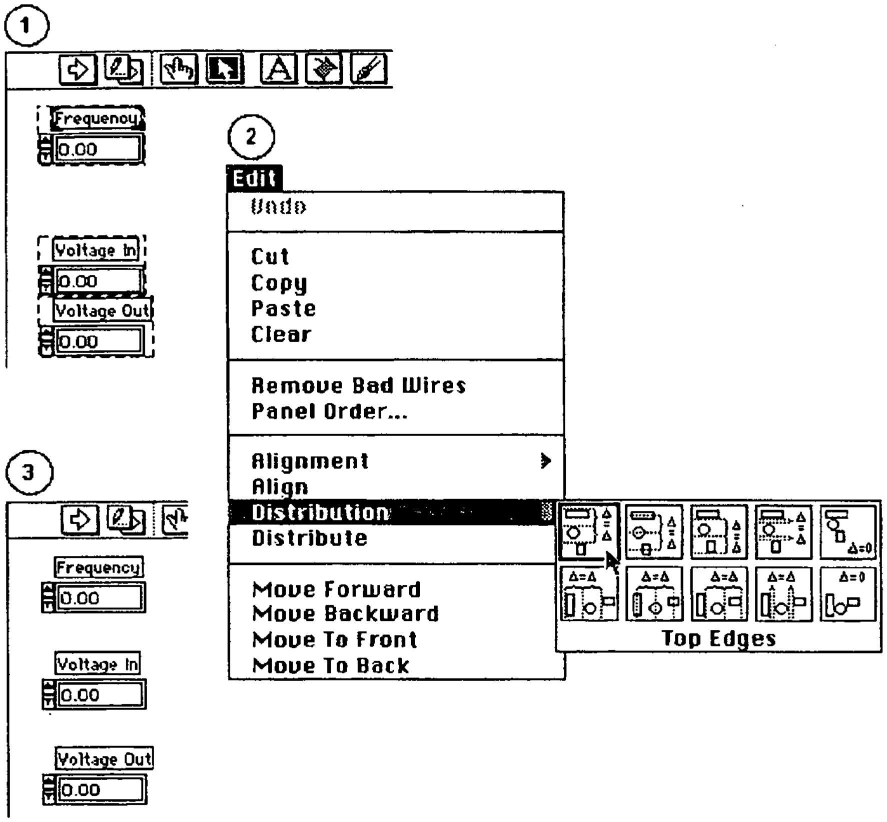
Select the objects you want to align and choose the axis along which you want to align them from **Alignment** in the **Edit** menu.



Choosing **Align** from the **Edit** menu uses the same axis for subsequent alignments.

Distributing Objects

To evenly distribute objects, select the objects you want to distribute and choose the axis along which you want to distribute them from **Distribution** in the **Edit** menu.

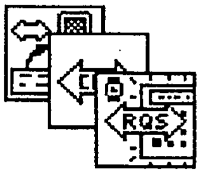


Choosing **Distribute** from the **Edit** menu uses the same axis for subsequent distributions.

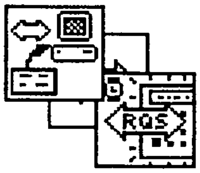
Moving Objects To Front, To Back, Forward, and Backward

You can place objects on top of other objects. LabVIEW has several commands in the **Edit** menu to move them relative to each other.

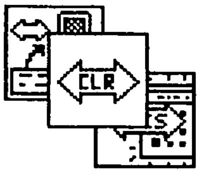
For example, assume you have three objects stacked on top of each other. Object 1 is on the bottom of the stack and object 3 is on top.



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.

Documenting VIs with the Get Info... Option

Selecting **Get Info...** from the **File** 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.)

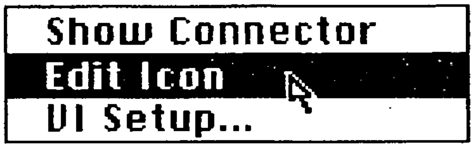
  The memory usage is divided into space required for the front panel and the block diagram, VI code, and data space. The memory usage can vary widely, especially as you edit and execute the VI. The block diagram usually requires the most memory. When you are not editing the diagram, save the VI and close the Diagram window to free space for more VIs. Saving and closing subVI panels also frees memory.

Icon and Connector

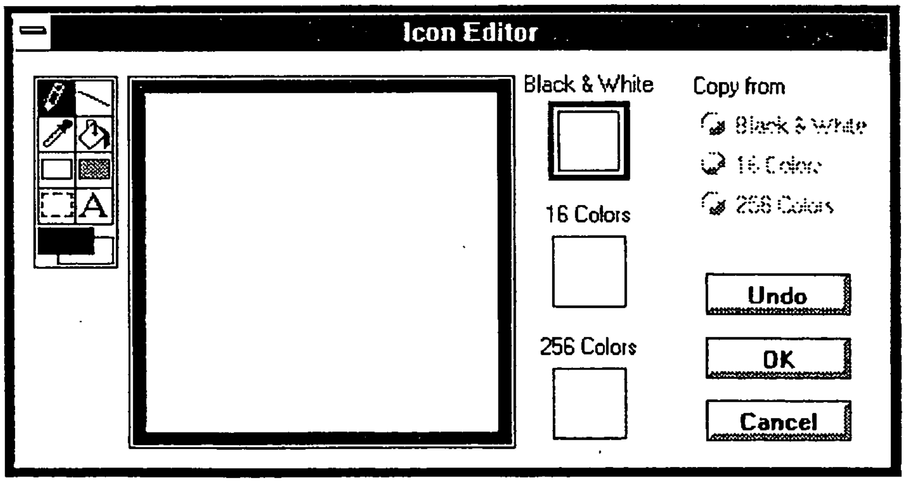
To call your VI from the block diagram of another VI, you must create an *icon* and *connector* for it. The icon is the graphical *name* of the VI, and the connector assigns controls and indicators to input and output VI terminals. This section explains how to create and edit a VI icon and connector.

Creating the Icon

To create an icon, make sure you are in edit mode. Either double-click on the icon, or pop up on the blank icon in the top right corner of the Panel window and select **Edit Icon**.



The following window appears.



You use the tools at the left of the window to create the icon design in the fat pixel editing area. The normal-size image of the icon appears in one of the boxes 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:** ***If you design a color icon only, it will not show up in a palette menu of the Functions menu if you place the VI in the*** `vi.lib` ***directory, nor will it be printed out or show up on a black and white monitor. LabVIEW uses the blank black and white icon in these cases, and the VI will appear to have a blank icon.***

The tool icons to the left of the editing area perform the following functions.

pencil
Draws and erases pixel by pixel. Use the shift key to restrict drawing to horizontal and vertical lines.

line
Draws straight lines. Use the shift key to restrict drawing to horizontal, vertical, and diagonal lines.

dropper
Selects the foreground color from an element in the icon. Use the shift key to select the background color with the dropper.

fill bucket
Fills an outlined area with the foreground color.

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

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

select
Selects an area of the icon for moving, cloning, or other changes. You can select an entire icon for copying and pasting into another VI with this tool.

A	text	Enters text into the icon. You can select a different font by double clicking on this tool icon.
■□	foreground/background	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> key will temporarily change all of the tools except the select tool to the dropper.

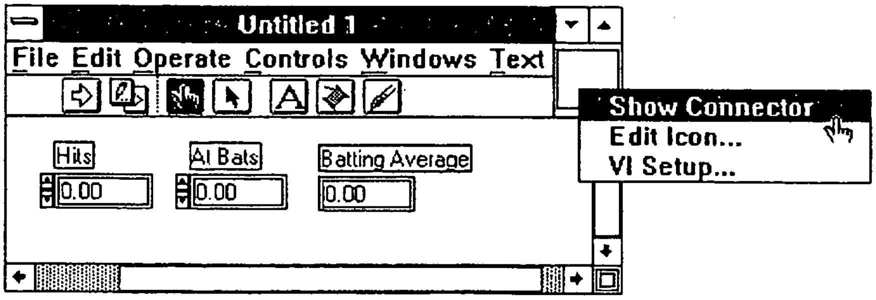
The buttons at the right of the editing screen perform the following functions when you click on them.

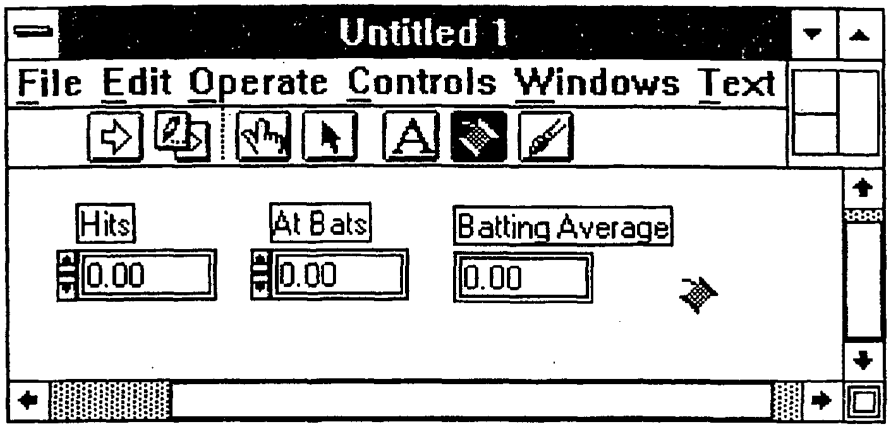
Undo	Cancels the last operation you performed.
OK	Saves your drawing as the VI icon and returns to the Panel window.
Cancel	Returns to the Panel window without saving any changes.

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 will use programmatically need terminals on the connector pane.

If the connector for your VI is not already displayed in the upper right corner of the Panel window, choose **Show Connector Pane** from the icon pane pop-up menu, as shown in the following illustration. The Diagram window does not have a connector pane.



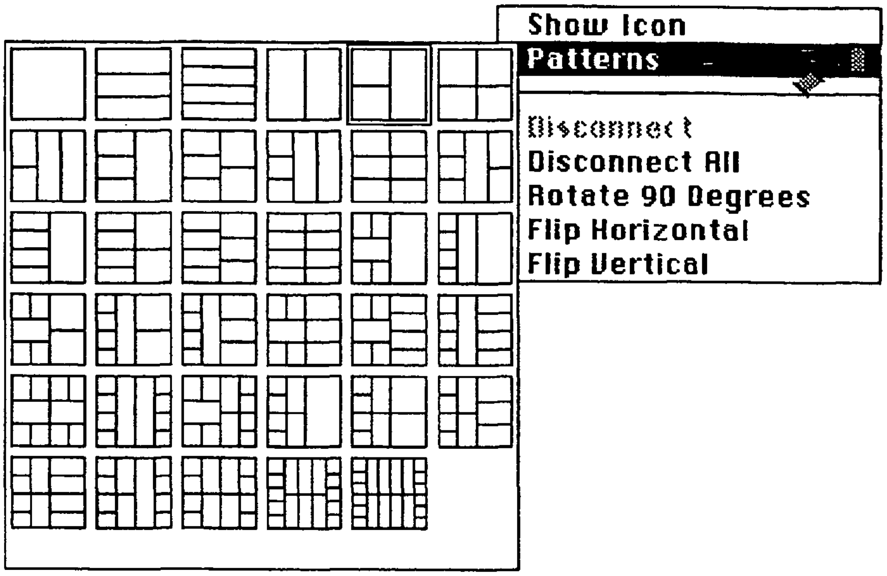


The connector replaces the icon in the upper right corner of the Panel window. LabVIEW tries to select a terminal pattern for your VI with as many terminals on the left of the connector pane as controls on the front panel, and as many terminals on the right of the connector pane as indicators on the 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.



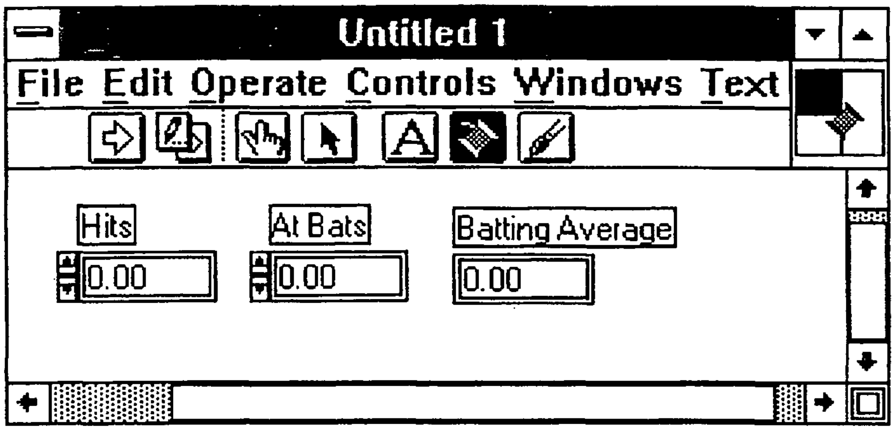
A solid border highlights the pattern currently associated with your icon, as shown in the previous illustration. To change the pattern, click on a new pattern. If you choose a new pattern, any assignment of controls and indicators to the terminals on the old connector pane is lost.

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.

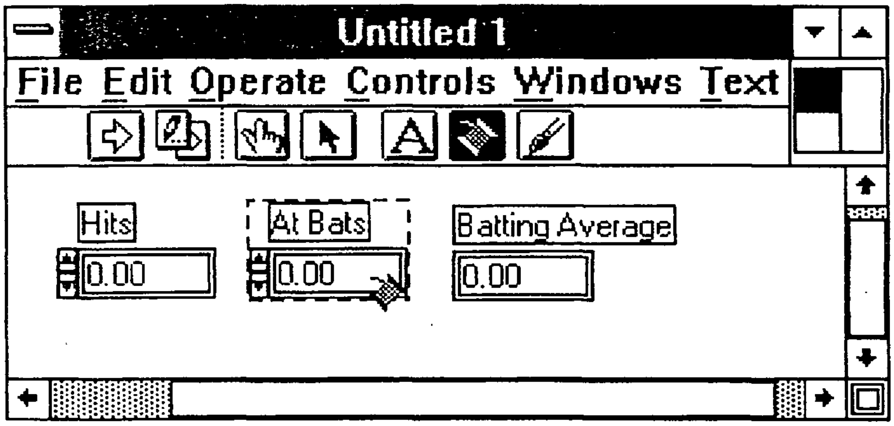
Assigning Terminals to Controls and Indicators

When you decide which terminal pattern to use for your connector, you need to assign front panel controls and indicators to the terminals. Follow these steps.

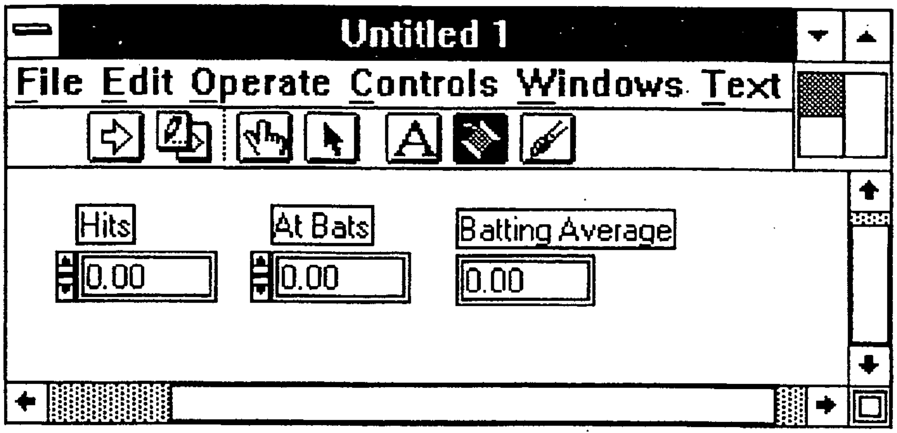
1. Click on a terminal of the connector. The tool automatically changes to the Wiring tool. The terminal turns black.



2. Click on the front panel control or indicator you want to assign to the selected terminal. A marquee frames the selected control.



If you position the cursor in free space and click, the dashed line disappears and the selected terminal dims, indicating that the control or indicator you selected now corresponds to the dimmed terminal.



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.***

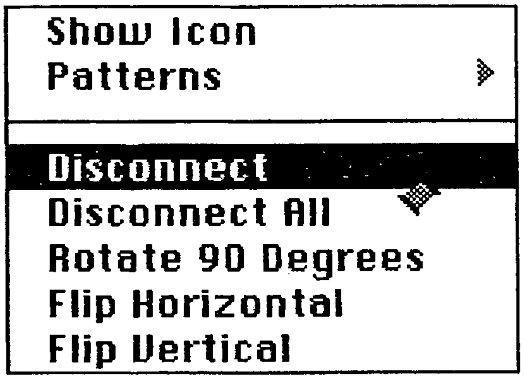
3. Repeat steps 1 and 2 for each control and indicator you want to connect.

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. 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, 20 terminals. If your front panel contains more than 20 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Deleting Connections

You can delete connections between terminals and their corresponding controls or indicators individually or all at once. To delete a particular connection, click on either the control/indicator or the terminal on the connector pane with the Wiring tool. Then choose **Disconnect** from the pop-up menu for the terminal you want to delete.



The terminal turns white, indicating that the selected connection no longer exists. To delete all connections on the connector, choose **Disconnect All** from the pop-up menu for the connector. LabVIEW also deletes all existing connections automatically if you select a new pattern from the **Patterns** palette.

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.

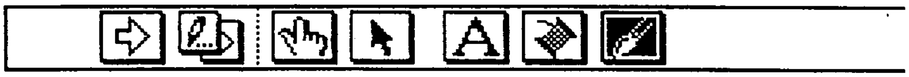
Chapter 3

Executing VIs

This chapter discusses how to operate and debug VIs, explains how to set up VIs and subVIs for special execution modes, and discusses factors that affect execution speed.

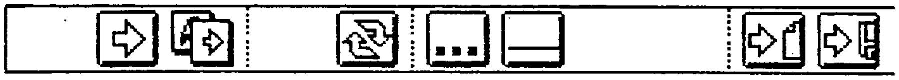
Operating VIs

LabVIEW has two modes of operation, edit mode and run mode. You edit and create VIs in edit mode, and you execute VIs in run mode. In edit mode, the editing tools are available on the Panel palette below the menu bar, as shown below.



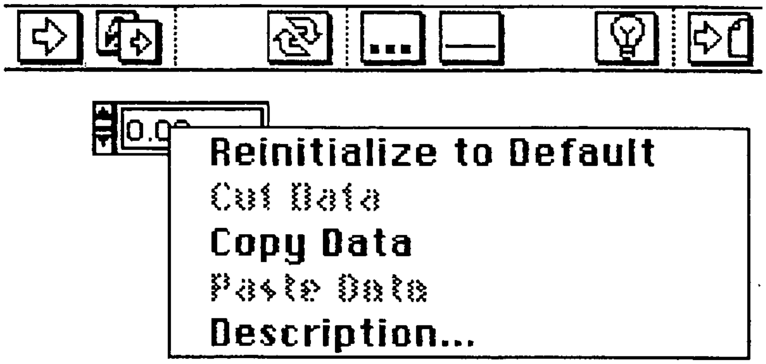
mode button

When you are ready to run the VI, click on the mode button, or select **Change to Run Mode** from the **Operate** menu. This compiles your VI and switches you to run mode. In run mode the execution buttons appear on the Panel palette, as shown below.



No tools are available in this palette. The Operating tool is the only tool that functions in run mode. You use the Operating tool to enter text into string controls, enter numbers into numeric controls, and manipulate mechanical-action controls. You cannot edit labels, or resize or move objects. You can only set inputs and select operating commands.

In edit mode, you use the tab key to change tools. In run mode, tabbing selects different objects on the front panel according to panel order (the order in which you placed them on the panel). In run mode, the pop-up menu contains only the options for running a VI. The pop-up menu for a control in edit mode contains a **Data Operations** submenu in addition to the editing submenus. Popping up on a control in run mode, however, displays only the **Data Operations** menu as shown in the following figure. You can cut, copy, or paste the contents of the control, set the control to its default value, and read the control's description with options in this menu. Some of the more complex controls have additional options, for example the array has options with which you can copy a range of values or go to the last element of the array.



The **Operate** menu on the menu bar contains commands for executing the current VI. The following sections discuss several fundamental execution-related tasks.

Running a VI

run button

You can run a VI in edit mode or run mode by selecting the **Run** command from the **Operate** menu or by clicking on the run button. From the edit mode, LabVIEW compiles the VI, switches to run mode and runs the VI. When the VI finishes executing, LabVIEW returns you to edit mode. While the VI is executing, the run button changes appearance.

VI running at top level

The VI is running at its top level if the run button changes to the VI running at top level indicator, as shown to the left.

VI caller is running

The VI is executing as a subVI if the run button changes to the VI caller is running indicator, as shown to the left.

You can run multiple VIs at the same time. After you start the first one, switch to the Panel or Diagram window of the next one and start it as described above. 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, refer to the *Debugging Techniques for Executable VIs* section of this chapter.

Stopping a VI

stop button

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 **Stop** command from the **Operate** menu. The **Stop** command aborts the top level VI at the earliest opportunity. The halted VI most probably 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, control 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 using the **VI Setup...** option from the icon pane pop-up menu on the VI's Panel window.

Running a VI Repeatedly

continuous run button

VI running continuously

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 run mode palette during continuous run is the same as during a single run started with the run button or the **Run** command.

Printing Programmatically

In designing VIs, you may want to print the results of a calculation or acquisition. You can print front panel data using the **Print...** option from the **File** menu. However, this option is not useful if you need programmatic control of printing, or if you want to print only a subset of the front panel data.

print mode button

If you set the print mode button of a VI, you can set a VI to automatically print the contents of its front panel each time it finishes execution. With this setup for top level VIs, you can automatically log to the printer the contents of a front panel after execution. For subVIs, the front panel prints at the end of each call.

If you want to print only at certain times, or if you want to print only a subset of the information on a front panel, create a subVI with controls for only the desired information, and set the print mode button for the subVI. When you want to print, call the subVI, pass it the desired data, and LabVIEW will print the data automatically.

For record keeping purposes, you can use the print button to print the front panel controls and indicators when the VI executes either at the top level or as a subVI. LabVIEW does not print the block diagram, icon, or connector when you use this feature, called *background print spooling*.

print mode active

When you click on the print button, it changes to indicate that printing is active. Click on this button at any time to disable printing.

Printing occurs when the VI completes and updates the indicators with new values. You do not see the standard **Print** dialog box, and LabVIEW uses the print options from the **Page Layout** dialog box. For minimum impact on execution performance, you should use background print spooling.

LabVIEW saves the printing state with the VI. If background print spooling is on in conjunction with a breakpoint suspension, LabVIEW prints the panel when you click the Resume button.

Front Panel Datalogging

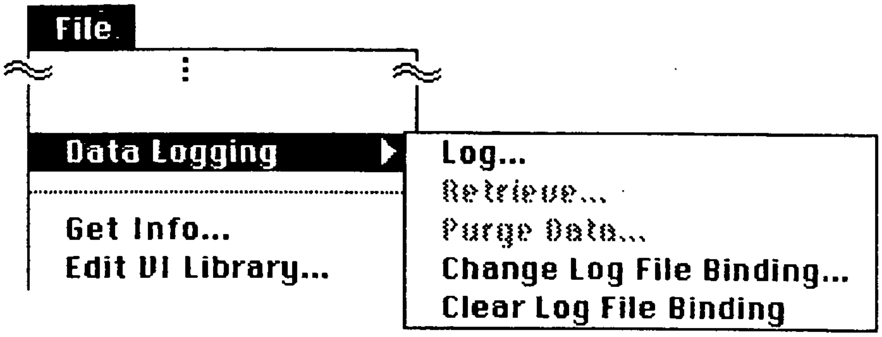
Front panel datalogging logs data to a separate file. This way you can have several separate files, each filled with logged data from different tests. You can also retrieve the data using standard file I/O functions.

datalogging button

datalogging activated

Turn automatic datalogging on for a VI by clicking on the datalogging button in the execution palette. When automatic datalogging is on, the arrow in the datalogging button darkens. Clicking on the activated datalogging button turns automatic datalogging off.

When automatic datalogging is on, the VI automatically logs front panel data to a file after each execution of the VI whether it is a top level VI or a subVI. You can also log the current front panel data at any time by selecting the **Log...** option from the **Data Logging** submenu of the **File** menu.

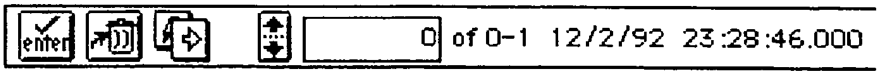


When datalogging is on, you will be prompted to select a file for the data, either immediately when you select the **Log...** option, or when the VI is run. After a file has been selected, the VI logs any further front panel data logged to this same file.

You can change the active file by using the **Change Log File Binding ...** option from the **Data Logging** submenu of the **File** menu.

Selecting **Clear Log File Binding** disassociates the VI from any associated log file. The next time you log from the VI front panel, you will be prompted to specify the log file.

To view logged data interactively, select the **Retrieve...** option from the **Data Logging** submenu of the **File** menu. The execution palette changes to the data retrieval palette, as shown in the following illustration.



enter button

delete button

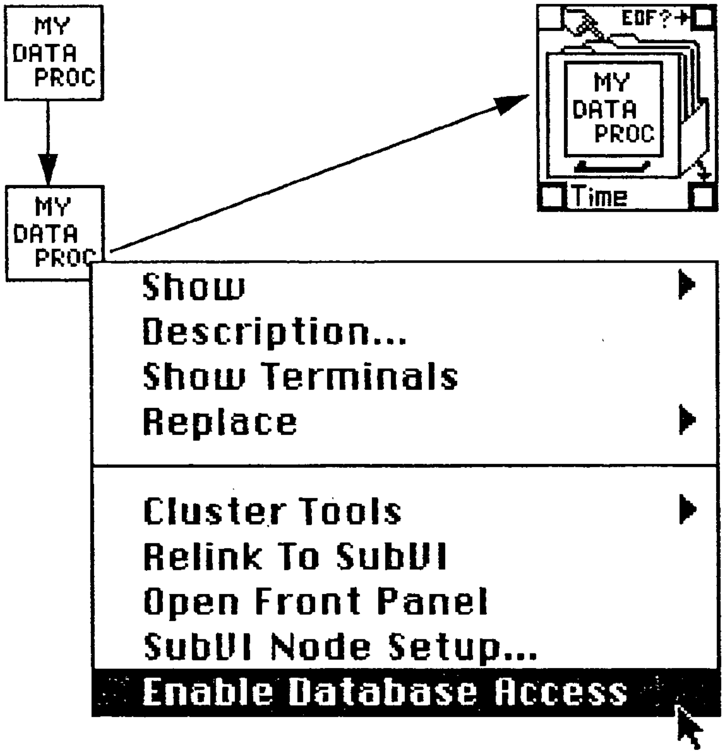
record marked for deletion

You change the record number to select a specific log entry. Clicking on the enter button causes the panel to display the selected record. You can mark the selected record for deletion by clicking on the delete button. When a 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 **Purge Data...** from the **File** menu, or until you switch out of data retrieval mode, either by clicking on the mode button or selecting **Retrieve...** from the **File** menu 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.

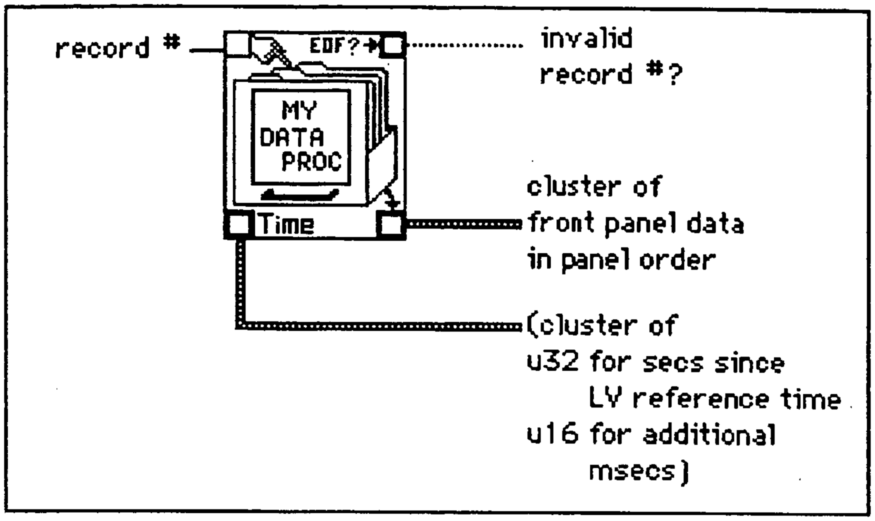
Programmatic Data Retrieval

You can retrieve data logged from a VI by using the standard File I/O functions to read the file as a datalog file.

You can also retrieve data by using the **Enable Database Access** option from the popup menu of a subVI that contains logged data. A *halo* that looks like a file cabinet will appear around the VI. This halo has terminals for accessing data from the VI. If you run the calling diagram, the subVI does not execute. Instead, LabVIEW retrieves data from a specified record. It also returns the time that the data was logged, and a Boolean value indicating whether the specified record is beyond the end of the file.



The following illustration shows the halo terminals for accessing the VI.


record #
EOF?
MY DATA PROC
Time
invalid record #?
cluster of front panel data in panel order
(cluster of u32 for secs since LV reference time u16 for additional msecs)

Debugging Executable VIs

The techniques discussed in this section apply only to VIs you can run. Broken (nonfunctional) VIs cannot compile or execute. See the *Debugging Techniques for Nonexecutable VIs* section in Chapter 12, *Wiring the Block Diagram*, for help in fixing broken VIs.

Correcting a VI Range Error

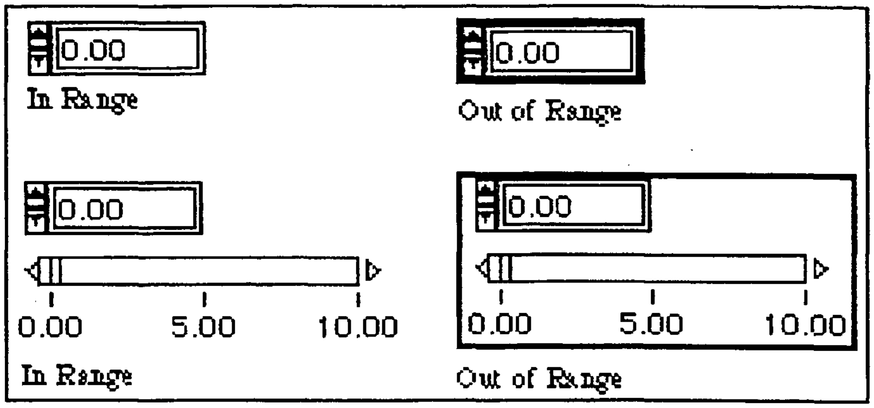
range error indicator

A range error indicator appears in place of the run button under the following circumstances.

- You configured a control on a subVI to stop execution when it receives an out-of-range value (via the control's **Data Range...** option), 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.

So that you can see the out-of-range control or indicator, its appearance changes as shown below.



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.***

Recognizing Undefined Data

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 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 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 will 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, here are some steps to take to solve the problem:

- Check wire paths to ensure that the wires connect to the proper terminals. Triple-clicking on the wire with the Positioning 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 **Windows** 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 if 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 *Using the Probe* section of this chapter 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 *Recognizing Undefined Data* section of this chapter.
- 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 *Recognizing Undefined Data* sections of this chapter.
- 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 the **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 like statements do in textual languages.

- Unlike functions, unwired subVIs do not generate errors while LabVIEW is in edit mode. If you mistakenly place one on the block diagram, the subVI will execute when the diagram does, degrading performance and perhaps corrupting the operation. You can inadvertently hide subVIs three ways: you can deposit one directly on top of another node; you can decrease the size of a structure without keeping the subVI in view; or you can place one off the main diagram area. For the latter case, scroll the Diagram window to its limits. Also check the inventory of subVIs used in the VI against the three **Windows** menu options (**[name of VI]'s Callers**, **[name of VI]'s SubVIs**, and **Unopened SubVIs**) to determine if any extraneous subVIs exist. The **Show VI Hierarchy** option can also help. You must move or resize nodes to search for extra nodes of a valid subVI if they exist.

LabVIEW Debugging Features

Single-Stepping through a VI

For debugging purposes, you may want to execute a block diagram node by node.

off on
step mode button

To enable single-step mode, click on the step mode button. The symbol changes from the flat line to the square wave. Click on this button at any time to return to normal execution.

step button

While the VI is running in single-step mode, the step button appears. Click on this button each time you want to execute the next node. When the VI finishes executing, the button disappears. If you return to normal execution mode before the VI completes, the step button disappears and the VI runs to completion at normal speed.

The step button affects 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.

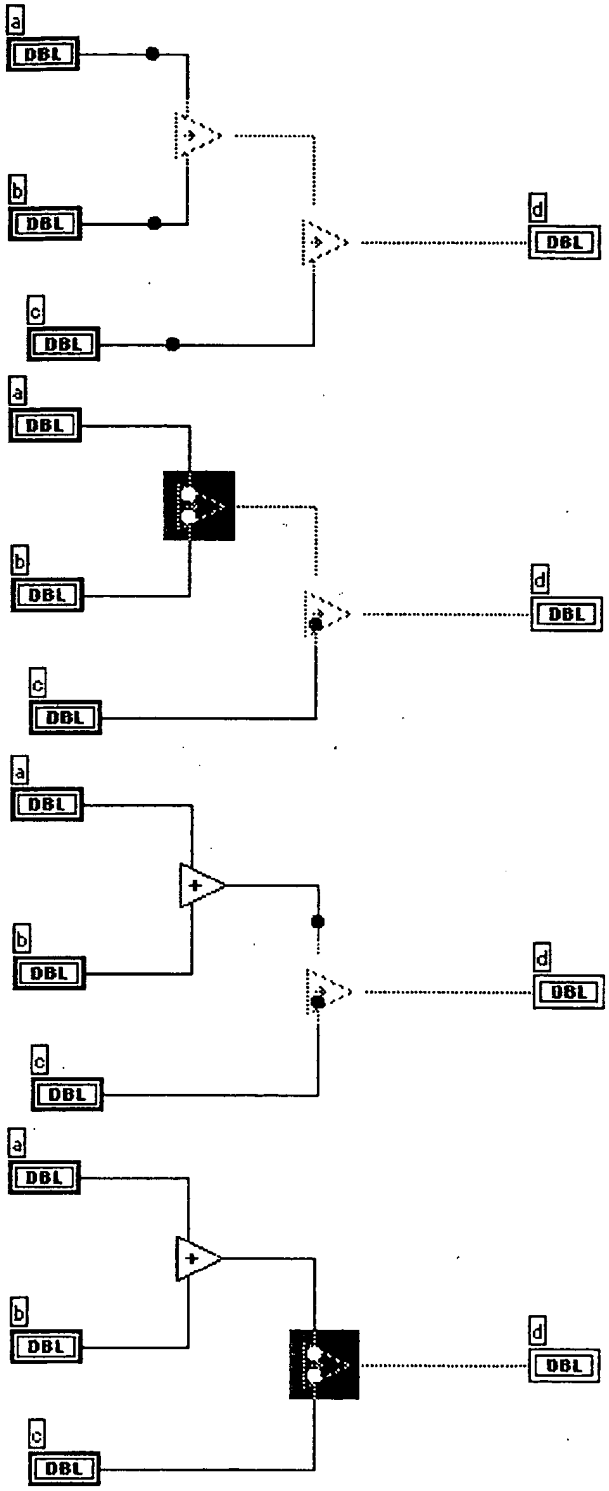
Highlighting Execution

For debugging purposes, you may want to view an animation of the execution of the VI block diagram.

off on
execution highlighting button

To enable this mode, click on the execution highlighting button, which changes appearance. Click on this button at any time to return to normal view mode. You commonly use execution highlighting in conjunction with single-step mode to gain an understanding of how data flows through nodes. Highlighting greatly reduces the performance of a VI.

As data passes from one node to another, the movement of data 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.

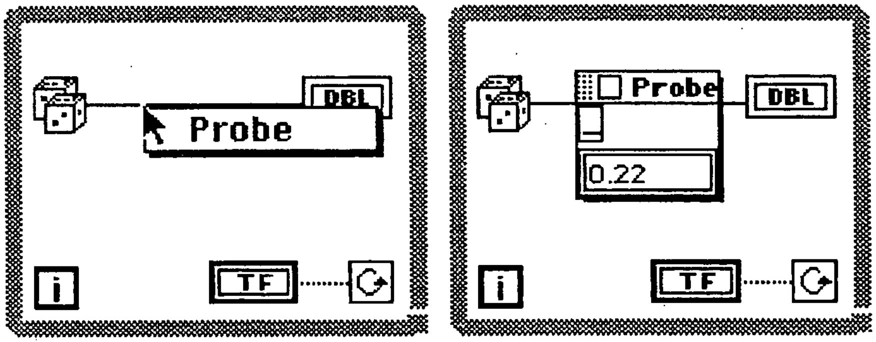

a
DBL
b
DBL
c
DBL
d
DBL
a
DBL
b
DBL
c
DBL
d
DBL
a
DBL
+
b
DBL
c
DBL
d
DBL
a
DBL
+
b
DBL
c
DBL
d
DBL

You can compile the VI without single-stepping or execution highlighting support code. This typically reduces memory requirements and increases performance by about 1 percent each. To do this, use the **VI Setup...** option from the icon pane pop-up menu on the Panel window and select the option to hide the buttons.

Using the Probe

The probe is a tool you can use to check intermediate values in a VI that executes but produces questionable or unexpected results. For instance, you may have a complicated diagram with a series of operations, any one of which may be returning incorrect data. You could wire an indicator to the output wire from one of the operations to display the intermediate results, or you can use the probe. 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.

The probe is similar to an indicator but is easier to use. The following example is a simple illustration of how to access the probe. In run mode, pop up on the wire leaving the Random Number (0-1) icon and select **Probe**. The probe display, which is a floating window, appears and flashes the values carried by the wire. You can use the probe in conjunction with execution highlighting and single-step mode to view values more easily. You must insert the probe before running your VI in order to see the data.



You cannot change data with the probe, and the probe has no effect on VI execution.

Setting Breakpoints

You may want to inspect the inputs to a subVI before it executes.

off on
breakpoint
button

To set a breakpoint, click on the breakpoint button as shown at left, which changes to a button with an exclamation point as shown. When that VI is called, it will halt execution of the higher-level VI so that you can inspect the subVI inputs. Click on this button at any time to remove the breakpoint and return to normal call mode.

You can also enable and disable the breakpoint with the **VI Setup...** option from the icon pane pop-up menu on the Panel window of the subVI. The two methods are interchangeable, and with either method the breakpoint occurs at all calls to the subVI. If a subVI is called from two locations in a block diagram, the breakpoint suspends execution at both calls.

If you want a breakpoint to suspend execution at a particular call to the subVI, set the breakpoint using the **SubVI Node Setup...** option from the subVI node pop-up menu, instead of the **VI Setup...** option. The **SubVI Node Setup...** suspends execution at that particular instance of the subVI only.

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 suspends as if it encountered a breakpoint. If no range error occurs, the subVI executes normally.

When a subVI encounters a breakpoint, its front panel opens or comes to the foreground and remains in a suspended state. 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 wish. In fact, you must change them to run the subVI if the range error indicator is on. The subVI's indicators display either default values or values from the last execution of the subVI during which its panel was open.

In the suspended state, two buttons appear on the subVI's run mode palette:

reset button

Click on the reset button, shown on the left, to run the subVI as if it were the top-level VI. You can also use the **Run** command in the **Operate** menu. 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 suspended state is the only time you can set values on indicators. You can rerun the subVI at the breakpoint as many times as you wish. While rerunning, you can return to the suspended state immediately by using the stop button or **Stop** command.

resume button

Click on the resume button, shown at the left, when you are ready to return the subVI's indicator values to the calling VI. If you click the Resume button without clicking the reset button first, the VI does not execute before returning.

▶ Note: ***You cannot set a breakpoint at a function or structure node, although you can encapsulate the function or structure in a subVI and suspend execution at the subVI call.***

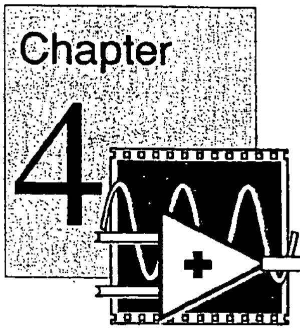
Warnings for Overlapping Objects

If an object is either partially or 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 box. You also get warnings if a wire has a loop in it (a place where the wire branches and then connects back into itself). Warnings do not prevent you from running a VI; they are just intended to help you to debug potential problems in your programs.

Front Panel Objects
Chapters 4 through 10

Front Panel Objects

Chapters 4 through 10



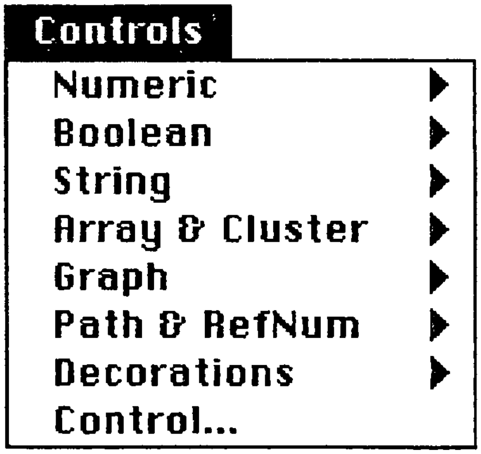
Introduction to Front Panel Objects

This chapter introduces the front panel and its two component parts, controls and indicators.

Building the Front Panel

Controls and indicators on the front panel are the interactive input and output terminals of the VI. This section explains a few editing options common to all controls and indicators.

The **Controls** menu on the front panel contains the following menu options.



- Numeric controls are used for entering and displaying numeric quantities.
- Boolean controls are used for entering and displaying Boolean (that is, True/False) values.
- String controls are used for entering and displaying text.
- Array and Cluster controls are used for grouping sets of data.

- Graph controls are used for plotting numeric data in chart or graph form.
- Decorations are pictures you can use to enhance the appearance of the front panel.
- **Control...** lets you choose a custom control of your own design.

When you select objects from the **Controls** menu, they appear on the Panel window. You can then move the controls with the Positioning tool, like any other LabVIEW object. Controls and indicators also have their own pop-up menus.

The **Decorations** item of the **Controls** menu offers a selection of graphics you can use to customize front panels. These objects are for decoration only, and do not display any data.

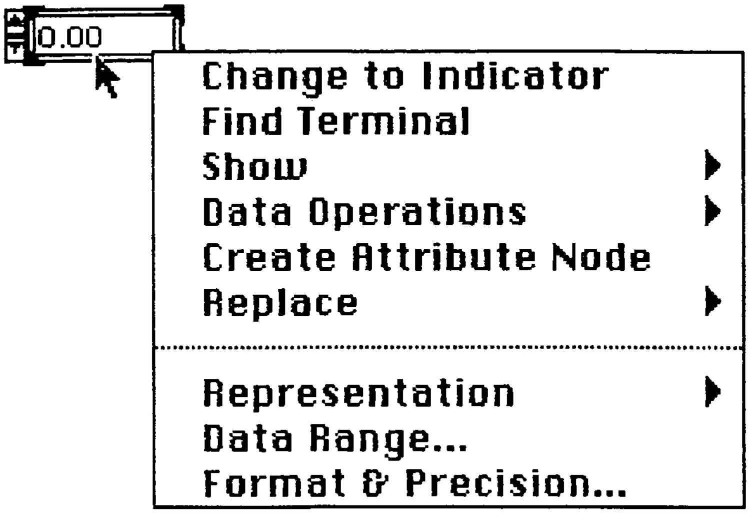
You can copy and paste controls and indicators from one front panel to another by using the **Copy**, **Cut**, and **Paste** commands from the **Edit** menu.

Importing Graphics from Other Programs

You can import graphics from other programs for use as static backgrounds or as states in Booleans or items in ring controls. Before you can use a picture in LabVIEW, you have to load it into the LabVIEW clipboard. You can use the **Import Picture** option from the **File** menu to import a graphics file into the LabVIEW clipboard. You can use **Import Picture** on GIF, PCX, BMP, TARGA, TIFF, LZW, WMF, and WPG files. To use the **Import Picture** option, you must have the file `lvimage.dll` installed on your system. If you copy an image directly from a paint program to the Windows clipboard and then switch to LabVIEW, LabVIEW automatically imports the bitmap to the LabVIEW clipboard. 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 from the control editor to import it as an image representing a value of a Boolean switch or a Pict Ring.

Common Front Panel Control and Indicator Options

When you pop up on a control or indicator on the front panel while in edit mode, 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 have only these options in their pop-up menus.



Objects in the **Controls** menu 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.

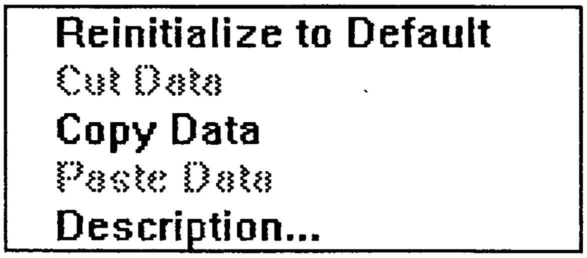
The **Find Terminal** item of the control and indicator pop-up menus highlights the block diagram terminal for the control or indicator. This option is useful for identifying a particular object on a crowded block diagram.

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.

In edit mode, 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. (In edit mode, you can paste data into an indicator.) Some of the more complex controls have additional options; for example, the array has options that allow you 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 when the VI is in run mode.



If you pop-up on a control while in run mode, you can only change the value of a control. In run mode you cannot change most characteristics of a control, such as its default value or description.

The **Replace** option of an object's pop-up menu displays the **Controls** menu palette from which you can choose a control or indicator to take the current item's place on the front panel. The item's label and description remain the same, as do its dataflow direction (control or indicator), numeric representation, scale configuration, mechanical action, and its position on the block diagram and connector pane. Wires also remain connected.

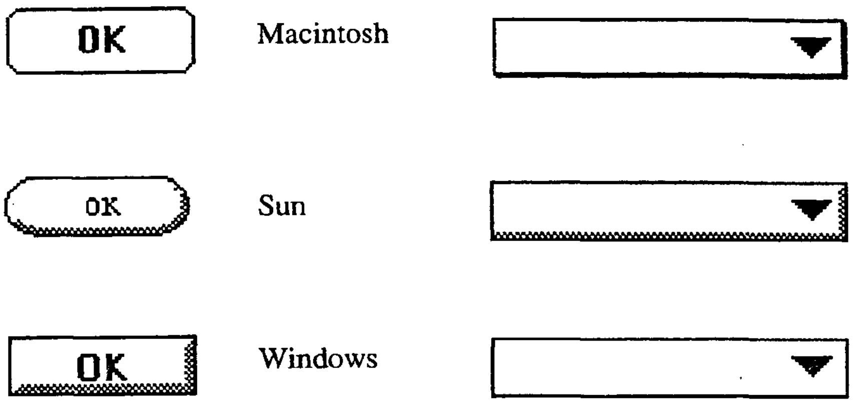
Dialog Controls

dialog box

dialog button

There are two types of dialog controls: dialog rings, which are numeric, and dialog buttons, which are Boolean.

The dialog controls change appearance depending upon which platform you are using. Each appears with the color and appearance typical of that platform. In other words, the dialog button ignores your color specifications and appears as a two dimensional black and white oval on the Macintosh, a three-dimensional oval gray button on the Sun, and a three-dimensional rectangular gray button in Windows. In the same way, the dialog ring appears as a black and white rectangle with a drop shadow on the Macintosh and as a flat black and white rectangle on the Sun and in Windows.



Because these controls change appearance, you can create VIs with controls that look at home on 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 will change 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.

Enter items into the dialog ring with the Labeling tool. Use <shift-enter> to go to the next item you want to enter. Pressing <enter> alone only results in a carriage return. Click outside the control or change tools to conclude entering terms.

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 control's name and icon appear in the **Controls** menu.

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 can automatically update all the VIs that use it.

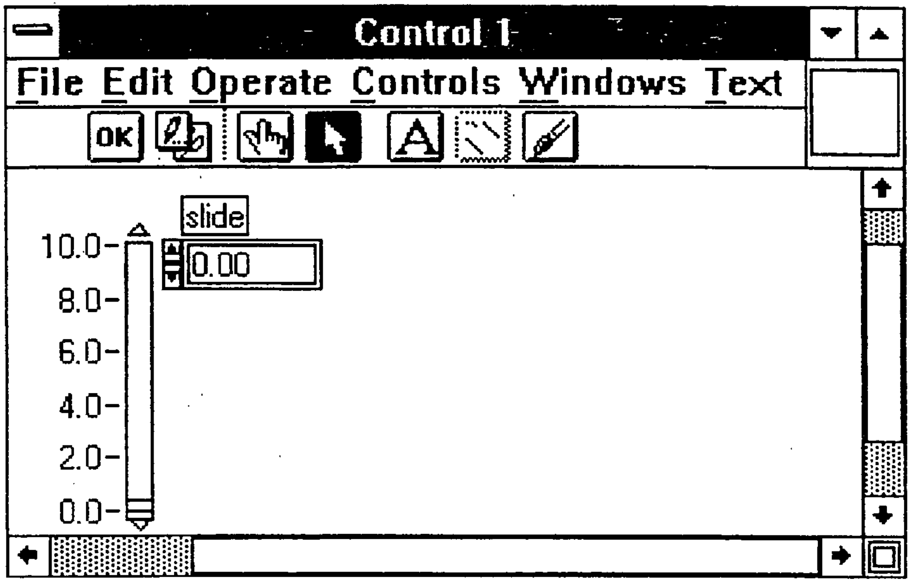
The following sections explain how to make these and other custom alterations to LabVIEW controls.

Creating a Custom Control

Put your VI in edit mode to customize a control. Place a control that is most like the one you want to create on the front panel. 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 Control** from the **Edit** menu. **Edit Control** 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.



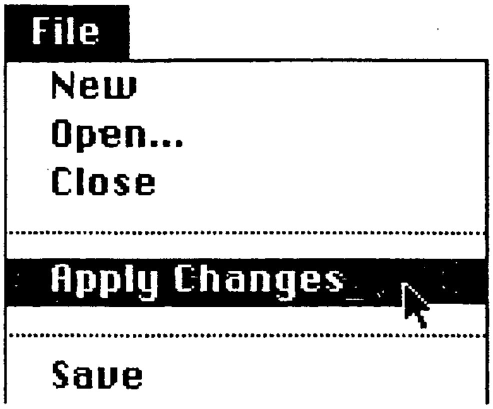
A control editor looks like a front panel, but it is used only for editing and saving a single control; it has no block diagram and cannot run.

A control editor has an edit mode and a customize mode. 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. In customize mode you can change the parts of a control individually. Customize mode is described in detail later in this chapter.

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

When want to replace the original front panel control with your new custom control select **Apply Changes** from the **File** menu of the control editor.



When you select **Apply Changes**, the following question appears: `Replace the original control <name> in <VI name> with Control N?` Click on the **Replace** button to replace the original control with your custom control. Click the **No** button to cancel the replacement.

If your original front panel is the only place you will 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 *Saving a Custom Control* section of this chapter.

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 **Open** from the File menu.

not-OK button

If the control editor has more than one control in it, the not-OK button appears. 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.

Saving a Custom Control

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 will ask you if you want to save the control.

Using a Custom Control

When you save your custom control, you can use it on other front panels by selecting **Control...** from the **Controls** menu 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.

Adding a Custom Control to the Controls Menu

For easier access to custom controls, save them in a directory or a VI library inside a `.lib` directory located in the LabVIEW library directory. By default, the LabVIEW library directory is the directory where LabVIEW is located.

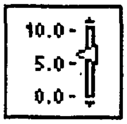
▶ Note: ***Do not save your controls in the*** `vi.lib` ***directory. This directory is updated by National Instruments as needed during new version releases of LabVIEW. Placing controls in*** `vi.lib` ***risks creating a conflict during future installations. Place controls in*** `.lib` ***directories in the LabVIEW library directory with, but not inside, the*** `vi.lib` ***directory.***

When you launch LabVIEW, LabVIEW looks at the contents of all `.lib` directories in the library directory. If any of the subdirectories or VI libraries contain controls, LabVIEW creates a palette for those controls and adds it to the **Controls** menu. You can then select your custom controls directly from the **Controls** menu.

You can have both custom controls and VIs inside your subdirectories or VI libraries. LabVIEW will display the controls on a palette accessible from the **Controls** menu of the front panel, and the VIs on a palette accessible from the **Functions** menu of the block diagram.

Your controls must be in a directory or a VI library within the `.lib` directory. LabVIEW will not display an individual control on a palette if it is saved directly into the `.lib` directory.

Making an Icon



control icon

If you save your controls so that they appear in the **Controls** menu, you should make an icon representing the control before you save it. Pop up or double click on the blank 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 **Open** from the File menu. A custom control will always open in a control editor window.

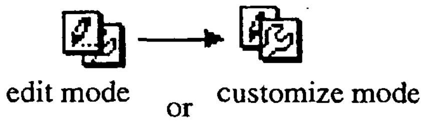
Changes you make to a custom control when you open it do not necessarily affect VIs that are already using that control. When you use a custom control on a front panel, there is no connection between that instance of the custom control to the file or VI library where it is saved; each instance is a separate, independent copy.

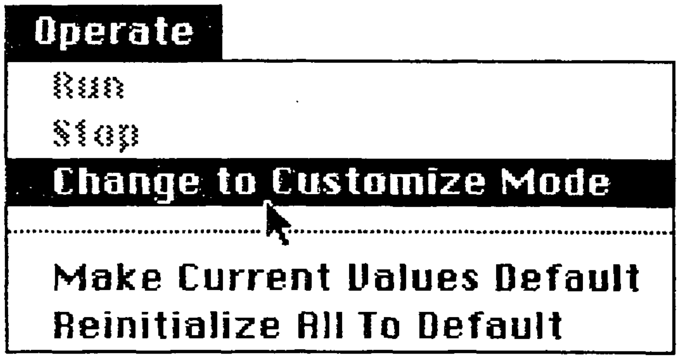
You can, however, create a connection between control instances on various VI front panels and the master copy of the control. To do this, you must save the custom control as a type definition or a strict type definition. Then, any changes you make to the master copy affect all instances of the control in all the VIs in which you use it. See the *Type Definition* section at the end of this chapter for more information.

Customize Mode

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, or by selecting **Change to Customize**

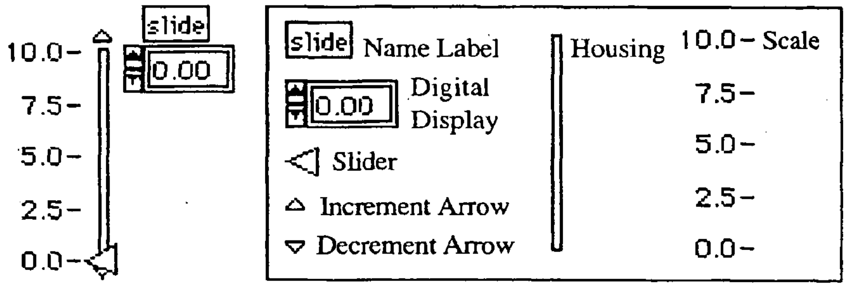
Mode or **Change to Edit Mode** from the Operate menu as shown in the following illustrations.





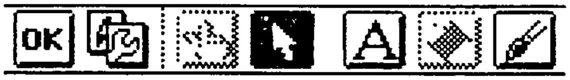
Independent Parts

All LabVIEW controls are built from smaller parts. 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 slide's scale with the placement tool, only the scale will move. You can select parts and align or distribute them using **Align** or **Distribute** from the **Edit** menu; 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 Operate tool is disabled.

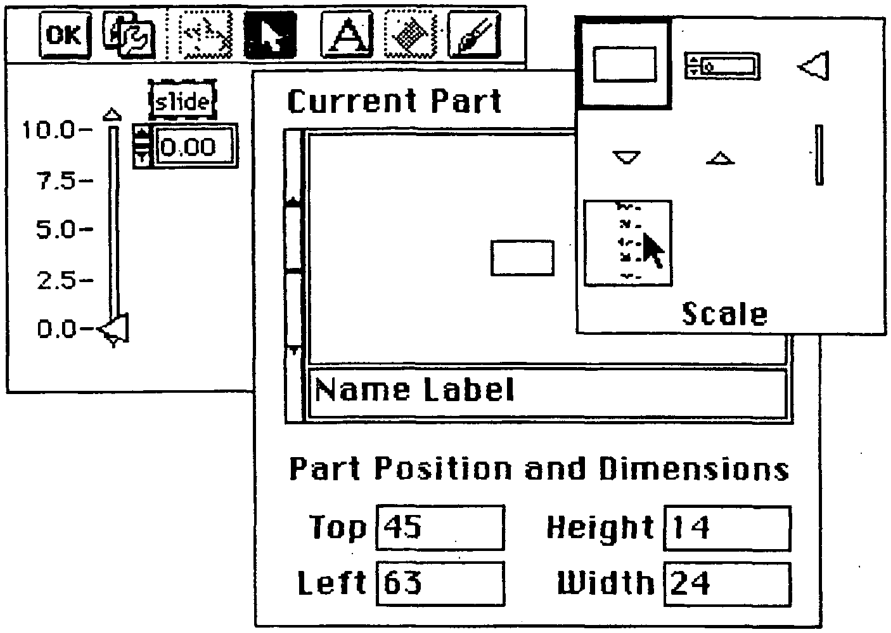


The wiring tool is always disabled in a control editor.

The Control Editor Parts Window

When you are in customize mode, you can select **Show Parts Window** from the **Windows** menu. 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 below shows the control editor window on the left overlaid by the parts window on the right. The slide's 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.



The parts window shows you the exact position and size of the part shown in the current part display. 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 illustration above the parts window displays the position and size of the slide's 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.

Pop-up Menus for Different Parts in Customize Mode

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, increment and decrement arrows, are the most common. Cosmetic parts show a picture.

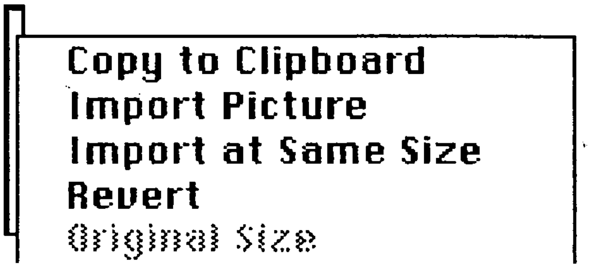
The second kind of part is a text part, such as the slide's name label. Text parts show a picture used as the background, usually just a rectangle, and some text.

Finally, a part may be another control. The slide, for instance, 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: the graph uses an array of clusters for its cursor display part, for instance.

The following sections describe the different parts and their pop-up options in more detail.

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 part's 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 correctly on both color and black and white monitors.

Finally, you can color pictures taken from LabVIEW parts or decoration with the LabVIEW coloring tool. Pictures imported from another source keep the colors they were imported with, because those colors are a part of the definition of that picture.

Import Picture and **Import at Same Size** replace a cosmetic part's current picture with whatever picture is on the clipboard. Use these options to individually customize the appearance of your controls, by importing pictures of an open and closed valve for a Boolean switch, for example.

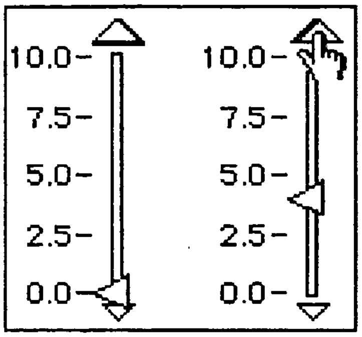
Import at Same Size replaces the current picture, but keeps the picture's old size, shrinking or enlarging the clipboard picture to fit. If there is no picture on the clipboard, both **Import Picture** and **Import at Same Size** are disabled.

Revert restores the part to its original appearance. **Revert** does not change the part's 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 **Open** from the File menu, **Revert** is disabled.

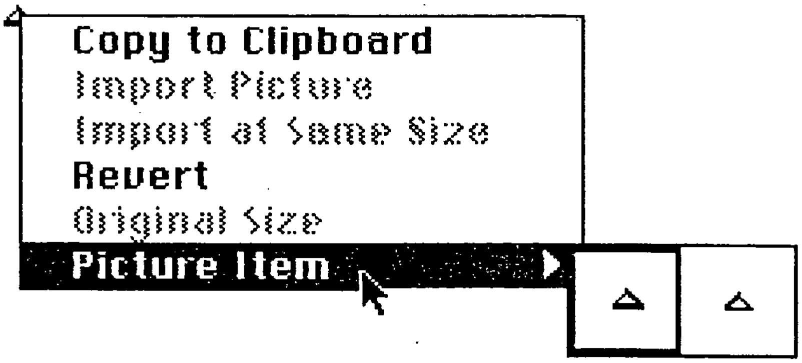
Original Size sets a part's picture to its original size. This is useful for pictures that you imported from other applications and then resized. Some of these pictures don't look as good as the original when resized, and you might want to restore their original size to fix them. If you have not imported a picture, **Original Size** is disabled.

Cosmetic Parts with More than One Picture

Some cosmetic parts have more than one picture, which they show at different times. These different pictures are all the same size and use the same colors. For example, the slide's increment arrow is a picture of a triangle, normally raised slightly from the background. It also has another picture, a recessed triangle, that shows while you are clicking on it with the Operate tool to increment the slide's value. The picture below shows the two pictures of an increment arrow in action.



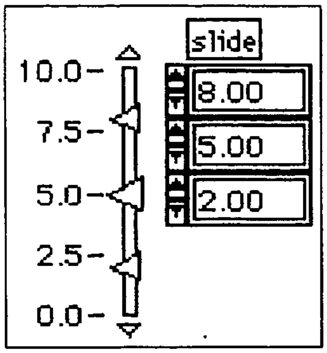
A cosmetic part with more than one picture has the Picture Item option on its pop-up menu, as shown below.



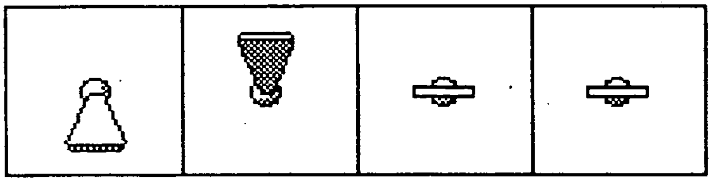
Picture Item shows all the pictures a cosmetic part has. The picture that is currently showing has a dark border around it. When you import a picture, you change only the current picture item. To import a picture for one of the other picture items, first select that picture item and then import the new picture.

Cosmetic Parts with Independent Pictures

A cosmetic part with more than one picture can have different sized pictures that each use different colors. The slide, for example, uses two different sized pictures 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.



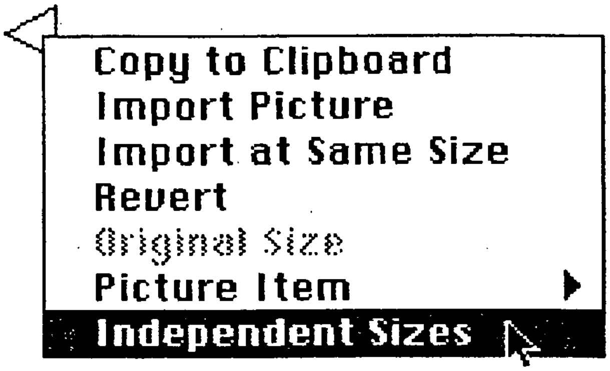
A Boolean switch also has more than one picture, and each picture may be a different size and have different colors. The first picture shows the False state; the second shows the True state; the third shows the transition state from True to False when the mechanical action **Switch/Latch When Released** is in effect; and the fourth shows the transition from False to True. In the illustration below, the last two pictures just happen to be the same.



1: False 2: True 3: T→F 4: F→T

You can also import pictures for a Boolean while in the control editor edit mode by using the edit-mode pop-up menu. You can import a picture for the False state, changing the first and third pictures, or for the True state, changing the second and fourth pictures.

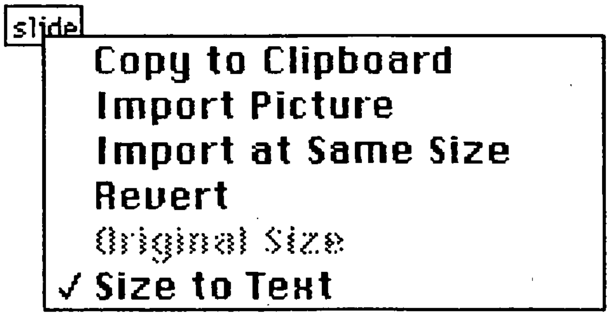
When a cosmetic part can have different sized pictures, the part has the **Independent Sizes** option on its pop-up menu, as shown below.



Independent Sizes is an option you can turn on if you want to move and resize each picture individually without changing the cosmetic part's other pictures. Normally, this option is not checked. When you move or resize the cosmetic part's current picture, its other pictures also move the same amount or change size proportionally.

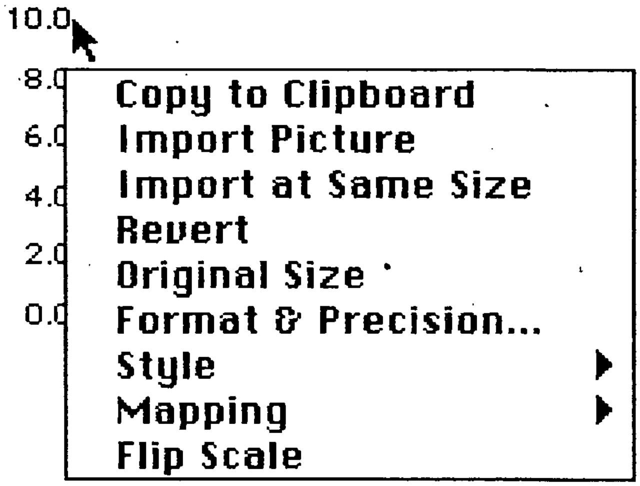
Text Parts

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 part's pop-up menu. The other items on this menu are the same as the text pop-up menu in front panel edit mode.



Scale Parts

A scale is a special kind of text part with markers for the text. The scale's 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.



Flip Scale 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. **Flip Scale** 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 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.

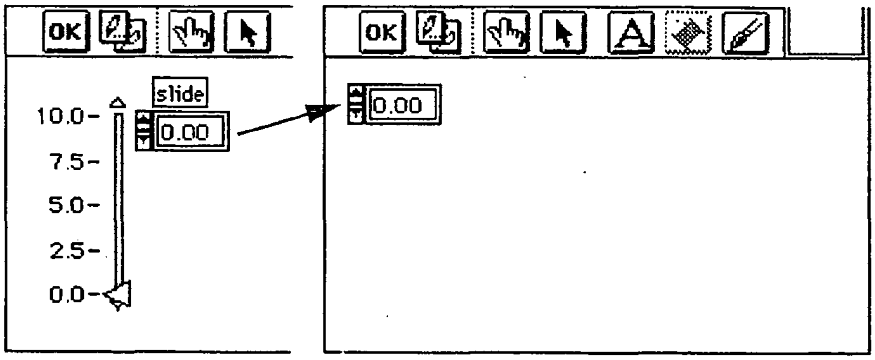
The digital display is also made up of parts. When you are editing the original control in the control editor, the digital display behaves as a single part, so you cannot change or move its parts individually. You can, however, open a control editor for the digital display and customize it there.

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 instance. Then you can choose **Edit Control** from the Edit menu.

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 figure 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.



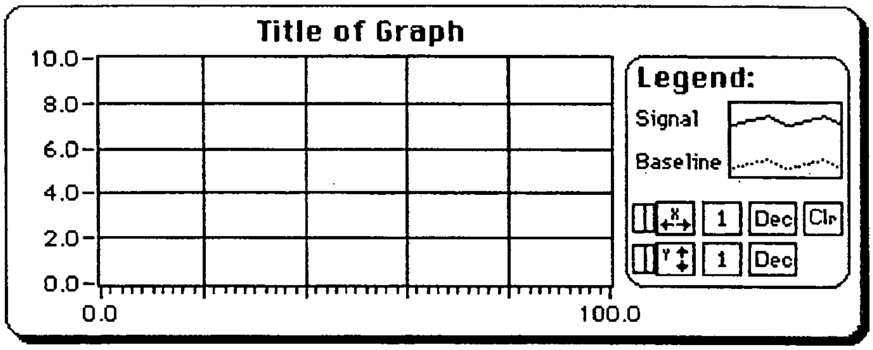
Adding Cosmetic Parts to a Custom Control

When you are making a custom control in the control editor, you can add cosmetic or text parts to it to make it even more distinctive.

If you paste a picture or text from the clipboard, create a label with the labeling tool, or select a picture from the **Decorations** menu, that picture or text becomes a part of your control and appears with the control when you place it on front panels. 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 illustration below 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.



Type Definitions

In addition to using the control editor to customize the appearance of a control, you can also use it to create a master copy of a control, called a *type definition*. Type definitions are useful where 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

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 will automatically update every VI that uses that type definition, or if you prefer, indicate 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 will update the type definition everywhere it is used.

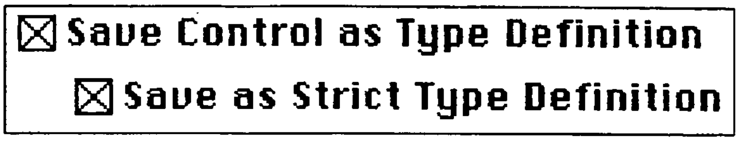
Strict Type Definition: Everything must match

A type definition can also force *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*.

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 will automatically update 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.

Making a Type Definition

You make a type definition by saving a control in a control editor window. Set up the control the way you want it, and choose **Save** from the **File** menu in the control editor window. Check the box marked **Save Control as Type Definition** in the dialog box before saving your control. To make a strict type definition, also check the box marked **Save as Strict Type Definition.**



☒ Save Control as Type Definition

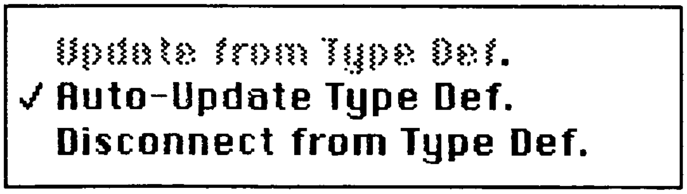
☒ Save as Strict Type Definition

You can open any type definition you have saved by selecting **Open** from the File menu. A type definition will always open in a control editor window. Any changes you make to a type definition affects all VIs that are using it.

Using a Type Definition

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. When you use a strict type definition on your front panel, however, you cannot edit it in any way, except to give it a name.

You can only tell that a control is a type definition when you see the type definition options in its pop-up menu, as shown below. You can easily 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*.

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 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 will have a broken Run arrow and the type definition on the front panel will be 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.

Searching for a Type Definition

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 will be disabled and the Run arrow will be 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 control's 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, use the Bundle by Name and Unbundle by Name functions on the block diagram to access the cluster's 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

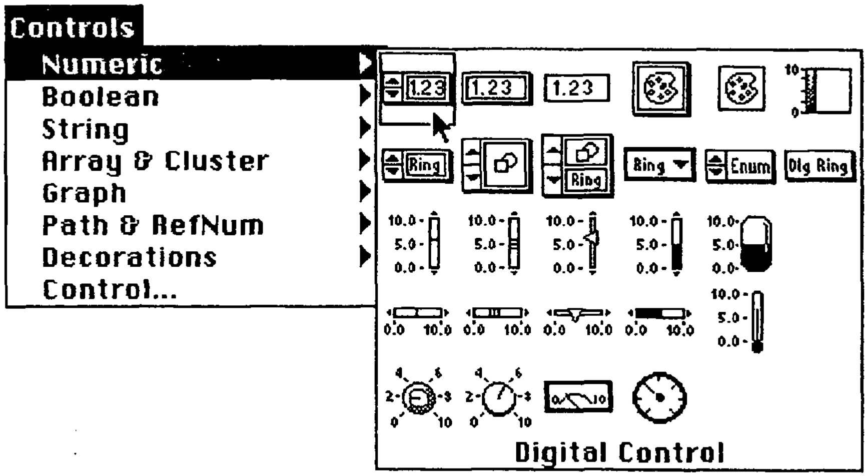
UnBundle by Name, you will have to change your block diagram. Refer to the description of these functions in Chapter 8, *Array and Cluster Controls and Indicators*, for more information.

Chapter 5

Numeric Controls and Indicators

This chapter explains how to edit and operate numeric controls and indicators. The first section introduces numeric objects, and subsequent sections describe the features of the different styles of numeric controls and indicators.

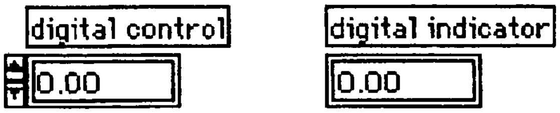
When you select **Numeric** from the **Controls** menu, a palette of controls and indicators appears. As you move the selection arrow over an object on the palette, a solid rectangular border surrounds the selected object, and the name of the object appears at the bottom of the palette.



In the above example, the digital control is selected. If you release the mouse button at this point, a digital control appears on the front panel, and a corresponding terminal will appear in the block diagram. Numeric controls and indicators are either digital, slide, rotary, ring, enumerated, color box, or color ramp controls. Block diagram terminals are described in Chapter 11, *Introduction to the Block Diagram.*

Digital Controls and Indicators

A digital numeric control and indicator are shown below.



Digital numerics are the simplest way to enter and display numeric data in LabVIEW.

Operating tool

You can click inside the digital display window with the Operating tool and then enter numbers from the keyboard, or you can click on the increment arrows with the Operating tool to increment or decrement the displayed value.

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 keyboard, 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 does not apply to values you change using the increment/decrement arrows.

Further, numeric controls accept only decimal digits, a decimal point, `+`, `-`, uppercase or lowercase `e`, and the terms `Inf` (infinity) and `NaN` (not a number). 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.

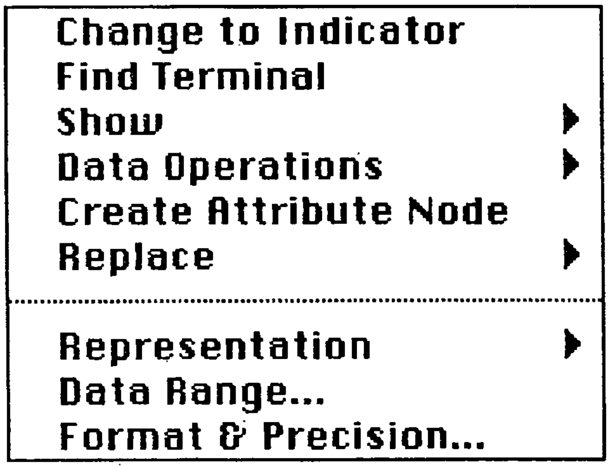
The increment buttons usually change the ones digit. Incrementing a digit beyond `9` or decrementing below `0` affects the appropriate adjacent digit. Incrementing and decrementing repeats automatically. If you click on an increment button and hold the mouse button down, the display increments or decrements 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.

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.

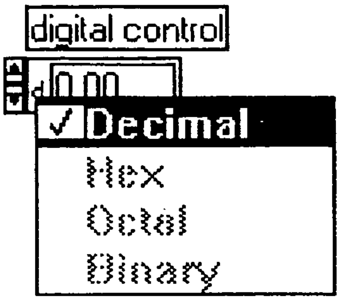
Digital Numeric Options

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.

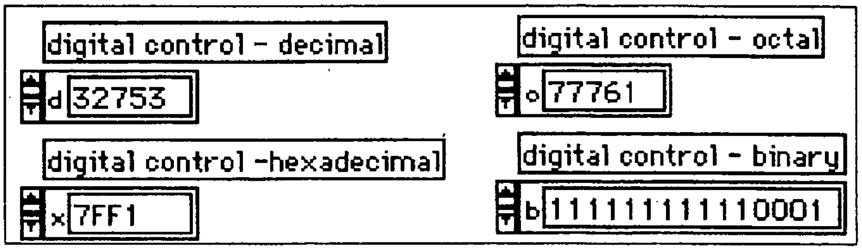


Displaying Integers in Other Radixes

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 **Show** submenu of 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.



digital control - decimal
d 32753

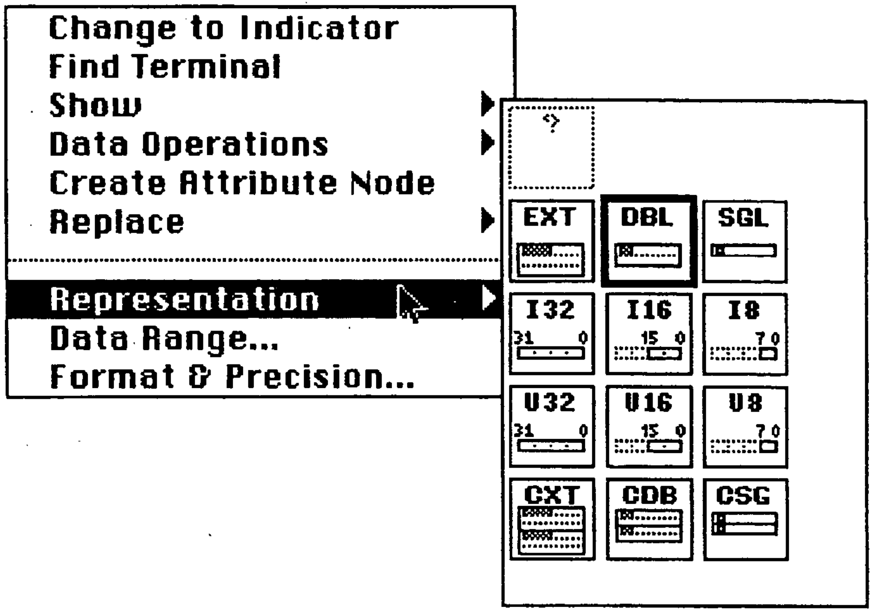
digital control -hexadecimal
x 7FF1

digital control - octal
o 77761

digital control - binary
b 111111111110001

Changing the Representation of Numeric Values

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), 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.

Setting the Range Options of Numeric Controls and Indicators

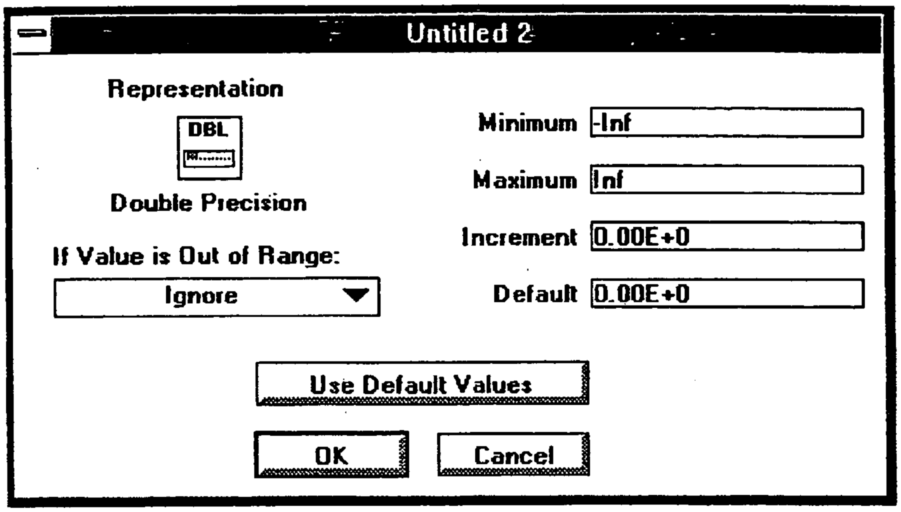
Each representation has natural minimum and maximum range limits. For example, signed byte integers are limited to values from -128 through 127, whereas floating-point numbers have the ranges shown in the following table.

Table 5-1. 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 above, 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.



Representation
DBL
Double Precision
If Value is Out of Range:
Ignore
Minimum -Inf
Maximum Inf
Increment 0.00E+0
Default 0.00E+0

Numeric Range Checking

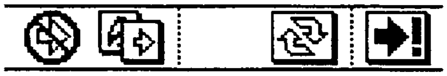
You can also limit intermediate values to certain increments. For example, you might limit word integers to increments of 10, or single-precision floating-point numbers to increments of 0.25. If you change either the limits or the increment, you should also decide what to do if a VI or the operator attempts 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 will not go beyond the minimum or maximum values.

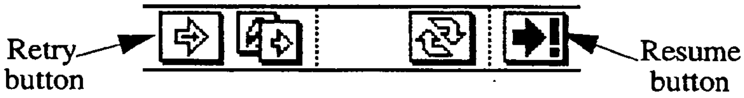
Coerce LabVIEW changes invalid values to the nearest valid value automatically. For example, if the minimum is 3, the maximum is 10, and the increment is 2, valid values are 3, 5, 7, 9, and 10. LabVIEW coerces the value 0 to 3, the value 6 to 7, and the value 100 to 10.

Suspend LabVIEW suspends execution of a VI or subVI when a value is invalid. However, VIs do not suspend in the middle of execution; they suspend only just before or just after a VI runs. 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.

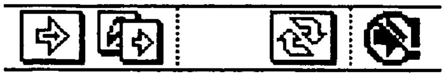
If the value of a control is invalid before a VI runs (either before you click on the Run button on a top-level VI, or when a subVI is about to execute), the VI is suspended. A suspended VI behaves as if there were a breakpoint at the beginning of that subVI. The front panel of the VI opens (or becomes the active window) and the invalid control(s) are outlined in red (or a thick black line in black and white). The run mode palette of a suspended subVI looks like the following illustration.



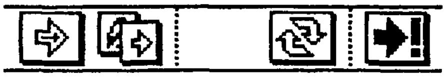
You must set the control to a valid value before you can proceed. When all control values are valid, the run mode palette looks like the following illustration, and you can click on the Retry button and then on the Resume button to continue execution.



If the value of an indicator is invalid when a subVI finishes running, or if an indicator was invalid at any time while the subVI was running, execution pauses as if there were a breakpoint at the end of that VI. The front panel of the VI opens (or becomes the active window) and the indicator(s) that are currently invalid are outlined in red (or a thick black line in black and white). The run mode palette of the suspended subVI looks like the following illustration.



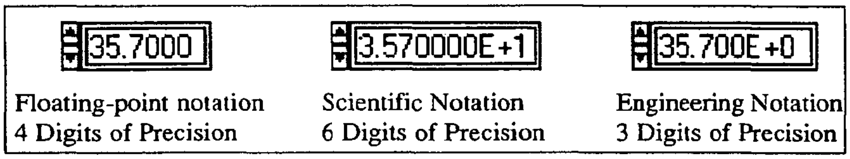
You must set the indicator to a valid value before you can proceed. In this situation, you can enter values directly into the indicator. You can also change the control values to produce valid outputs and run the subVI again by clicking the Retry button. When all indicator values are valid, the run mode palette looks like the following illustration, and you can click on the Resume button to continue execution.



Changing the Format and Precision of Digital Displays

In their default settings, digital displays have two digits of fractional precision. That is, they display two digits to the right of the decimal point. You can change the precision for displays to be 0 through 20 digits. The precision you select affects only the display of the value; the internal accuracy still depends on the representation.

You can change the notation from decimal to scientific or engineering as shown in the following illustration.



35.7000

Floating-point notation
4 Digits of Precision

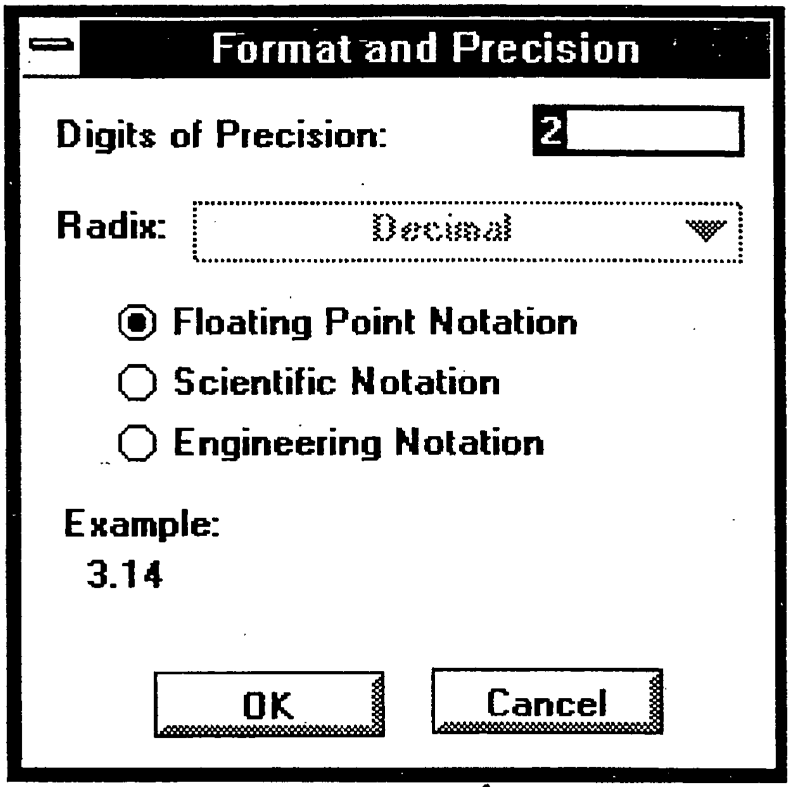
3.570000E+1

Scientific Notation
6 Digits of Precision

35.700E+0

Engineering Notation
3 Digits of Precision

To change either of these parameters of the digital display, use the dialog box shown in the following illustration, which appears when you select the **Format & Precision...** option from the display pop-up menu.



Format and Precision

Digits of Precision: 2

Radix: Decimal

● Floating Point Notation

○ Scientific Notation

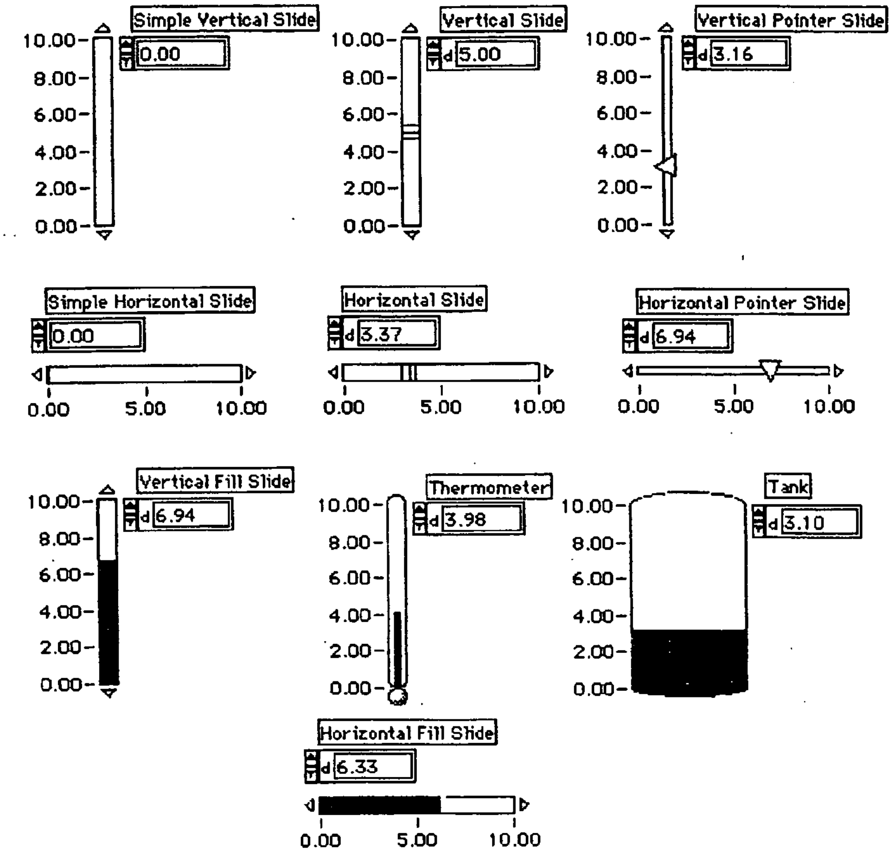
○ Engineering Notation

Example:
3.14

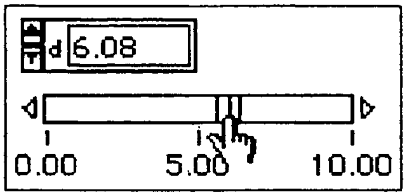
The display updates after you click in the OK button.

Slide Numeric Controls and Indicators

The slide controls and indicators are shown in the following illustration.

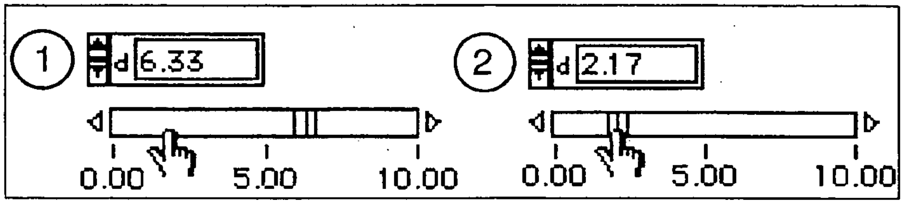


Each slide has a digital display. You can use the digital displays to enter data into slide controls, as explained in the *Digital Controls and Indicators* section of this chapter. You can use the Operating tool on such parts as the slider, the slide housing, the scale, and increment buttons to enter 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.

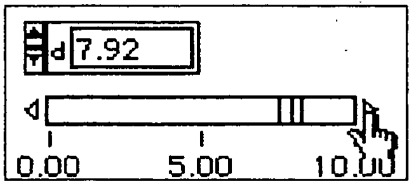


You can drag the slider with the Operating tool to a new position. If the VI is running during the change, intermediate values may pass to the program, depending on how often the VI reads the control.

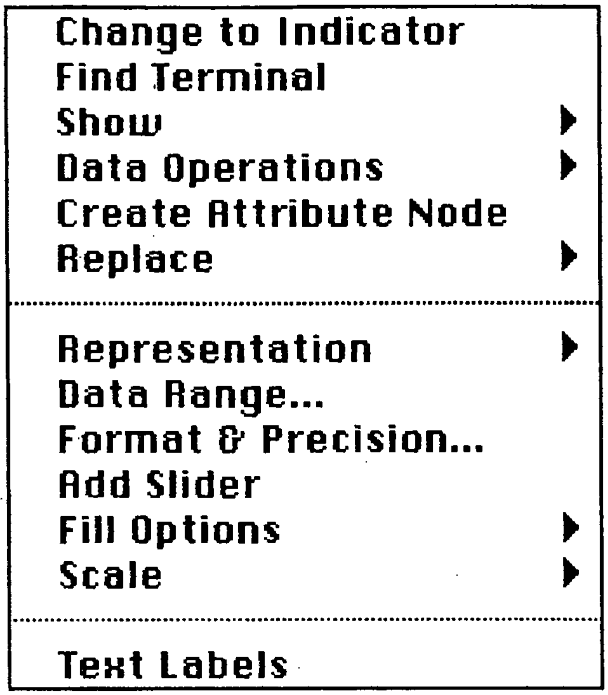
You also can click on a point on the housing and the slider will snap to that location as shown in the following illustrations. Intermediate values will not pass to the program.



If you use a slide that has increment buttons, you can click on an increment button, as shown below, and the slider will move slowly in the direction of the arrow. Hold down the shift key to move the slider faster. Intermediate values may pass to the program.



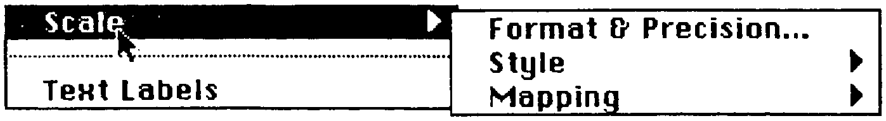
Just like the digital numerics, slides have **Representation, Data Range...**, and **Format & Precision** options in their pop-up menus. These options work the same as they do for digital displays, except that slides cannot represent complex numbers. Slides also have other options. The following illustration shows a slide pop-up menu.



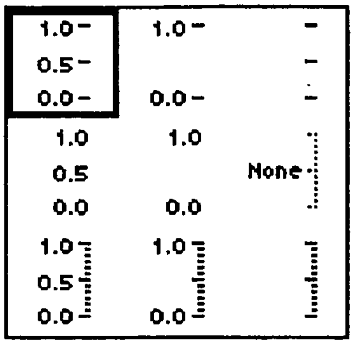
The **Digital Display** option in the **Show** submenu of the pop-up menu controls whether the slide's digital display appears.

Slide Scale

The scale submenu options apply to the slide's scale only. The **Format & Precision** option functions as described in the *Changing the Format and Precision of Digital Displays* section of this chapter. The scale pop-up menu is shown in the following illustration.



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.

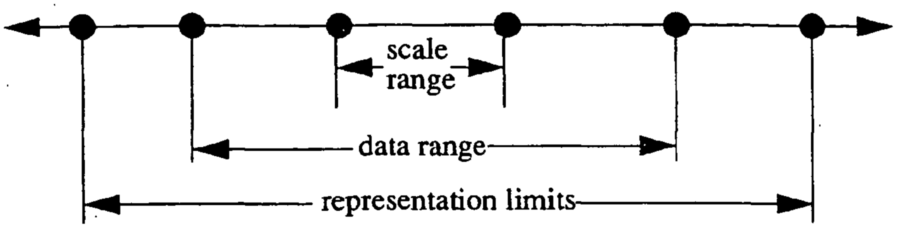


The **Mapping** item gives you the option of linear or logarithmic scale spacing, as shown in the following illustration.

If you change to Logarithmic spacing, and the low scale limit is less than or equal to zero, the limit automatically becomes 1 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.

Changing Scale Limits

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. 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.



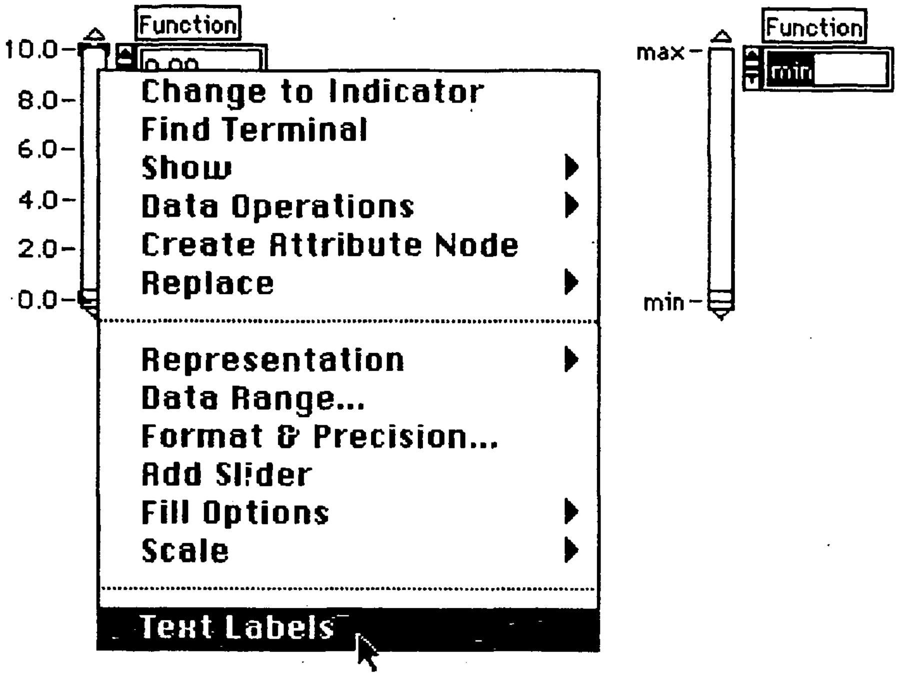
You can change a scale's minimum, maximum, and increment in five ways using either the Operating tool or the Labeling tool.

- 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.

Text Scale

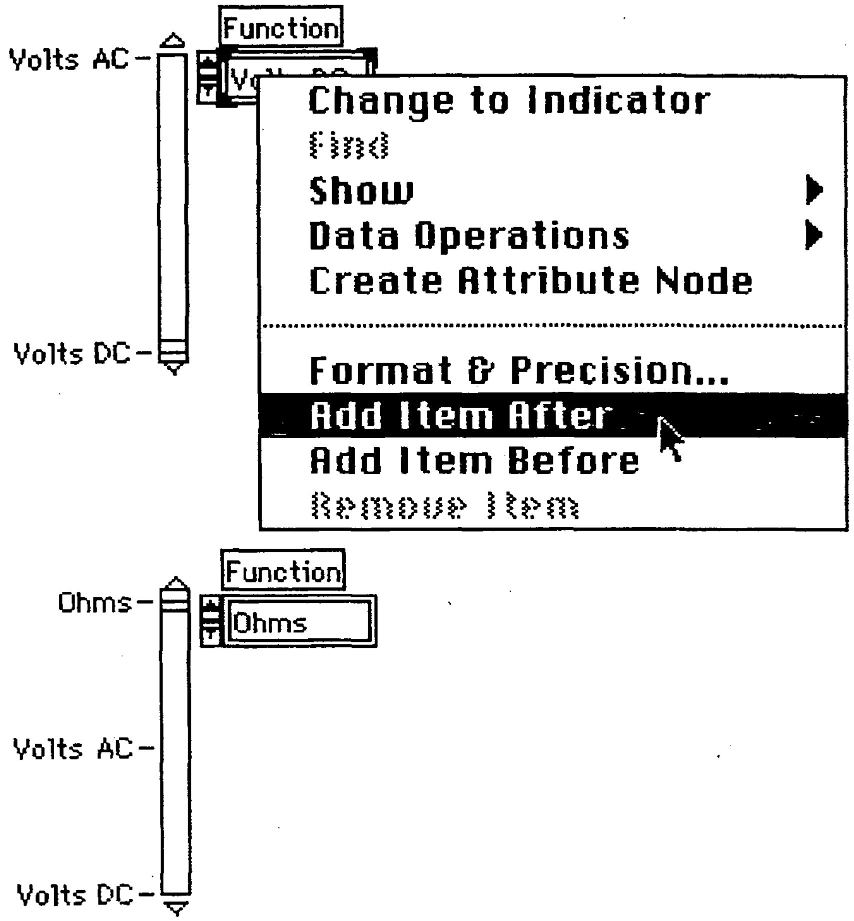
You can also use text labels on numeric scales. Labels are useful because they allow you to associate unsigned 16-bit integers with strings. This configuration is useful for selecting mutually exclusive options. To enable this option, select **Text Labels** from the slide pop-up menu. The slide control appears with default text labels, `min` and `max`, and you can begin typing immediately to enter the labels you want. Press <shift-enter> after each label to create a new label, and enter 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 **Digital Display** from the **Show** submenu to find out what numeric values are associated with the text labels you create. These values always start at 0 and increase by 1 for each text label.

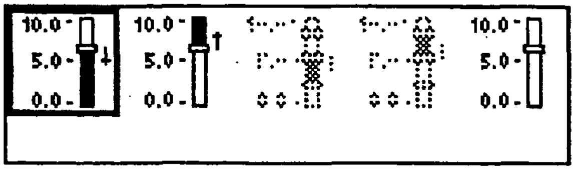
Use the **Add Item After** or **Add Item Before** option from the text display pop-up menu to create new labels, as shown in the following illustration.



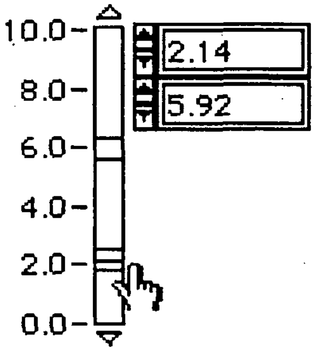
You can also press <shift-enter> to advance to a new item when you are editing the existing items.

Filled and Multivalued Slides

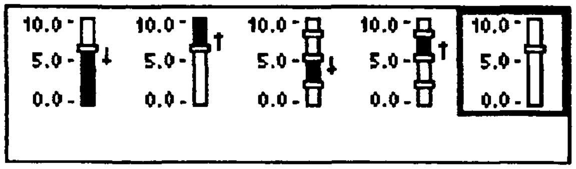
The **Numeric** palette contains four controls that are configured to fill from the minimum to the slider value. These controls include a vertical slide, a horizontal slide, the tank, and the thermometer. Using the **Fill Options** item in the pop-up menu, you can turn all slides into fill slide, and you can turn fill slides into regular slides. Normally, you have three choices, as shown in the following illustration: 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.

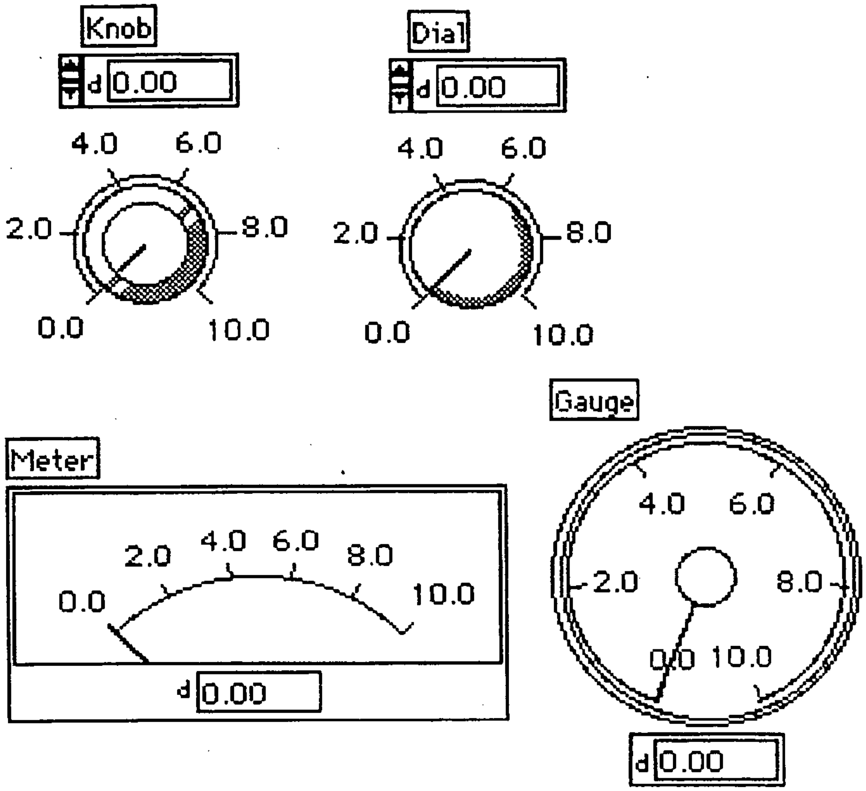


When you do this, two more options appear in the **Fill Options** palette: **Fill to Value Above** and **Fill to Value Below**. These options apply to the active slider.

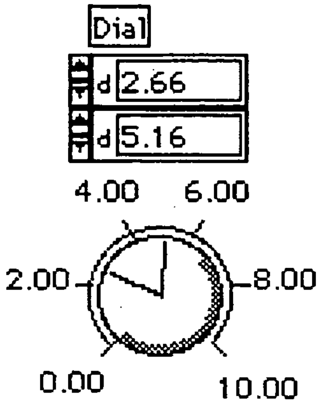


The Rotary Numerics

The rotary numerics are shown in the following illustration.



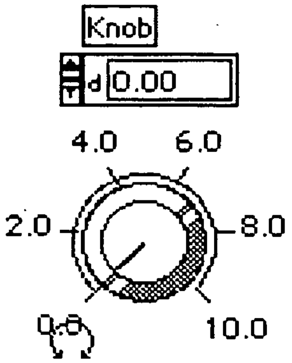
The rotary objects have all the same options as the slide, except for the fill options. The sliders (or needles) in rotary objects turn rather than slide, but you operate them the same way you operate slides.



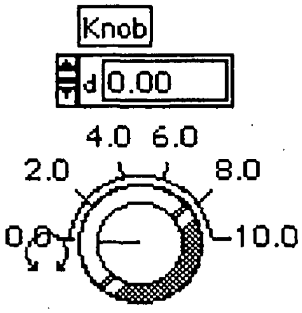
Rotary controls, like linear controls, can display more than one value, as shown in the illustration above. You add new values by selecting **Add Needle** from the pop-up menu, in the same way you add new values to slides.

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 will still have 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.

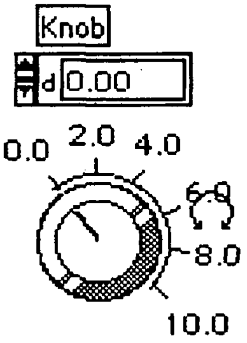
Place the cursor on the knob scale, where it changes appearance to a double arrow horseshoe.



Dragging outer markers changes the size of the scale arc.

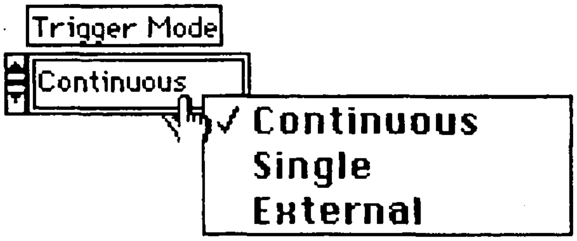


Dragging inner markers rotates the arc.



Ring Controls

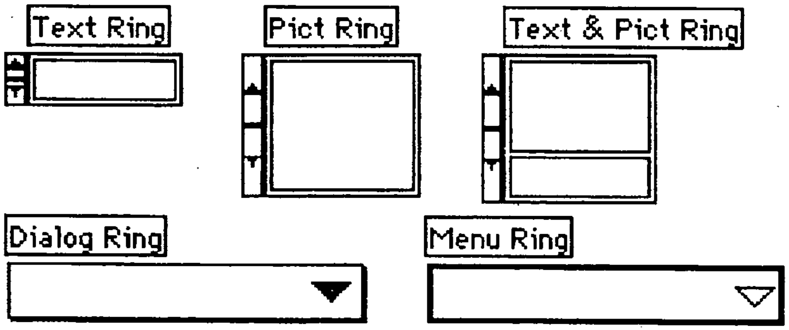
Rings are special numeric objects that associate unsigned 16-bit integers with strings, pictures, or both. They are particularly useful for selecting mutually exclusive options, such as trigger modes. For instance, you may want users to be able to choose from continuous, single, and external triggering.



In the example above, 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 will display the last item (External, above) for any value ≥ 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. The rings are shown in the following illustration.



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 display window as you do with labels, using the Labeling tool. Press the keyboard enter key or click outside the display window to end the editing process. The **Add Item Before** inserts a new item in front of the current item. The **Add Item After** option from the text ring pop-up menu creates a new empty item following the current one. 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> after typing in an item to advance to a new item.

You add items to the menu ring the same way you add items to the text rings. 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.

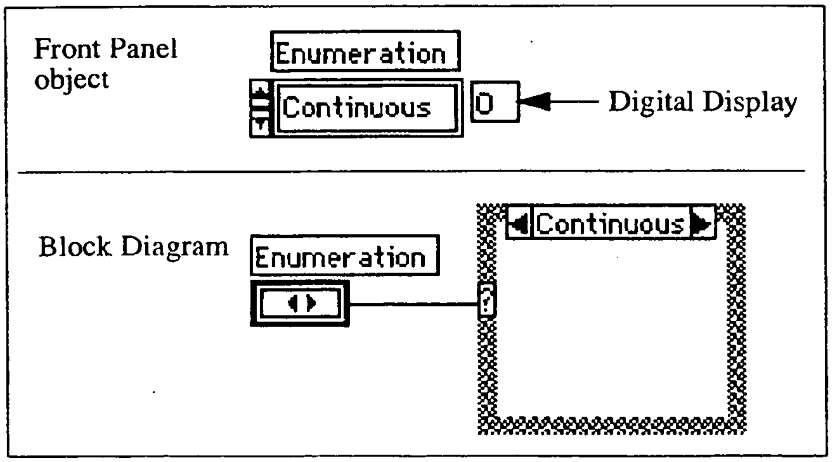
For every ring, the item values above the insertion point increase by one to adjust for the new item. For example, if you insert an item after item 4, the new item becomes item 5, the previous item 5 becomes item 6, 6 becomes 7, and so on. If you insert an item before item 4, the new item becomes item 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 item numeric values adjust automatically.

If you set a ring indicator or control to a value smaller than 0 or greater than the number of items included, the control or indicator displays the first or the last item, respectively.

Enumerations

An enumeration is similar to a ring control. If data from an enumeration is connected to a case structure, however, the case displays the string instead of the number. The enumeration data type is unsigned byte, unsigned word, or unsigned long, selectable from the **Representation** palette.



To enter items into an enumeration, use the Labeling tool. Press <shift>-<enter> to enter a new item. Click outside the enumeration when you have finished entering items.

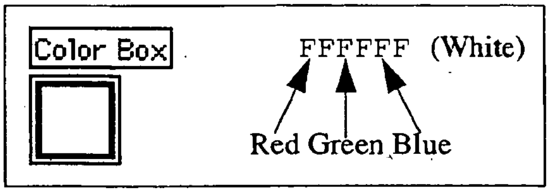
If you enter more than two items into an enumeration, you must add cases to the case structure for all the conditions to be available as cases.

All arithmetic operations except Add One and Subtract One treat the enumeration 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.

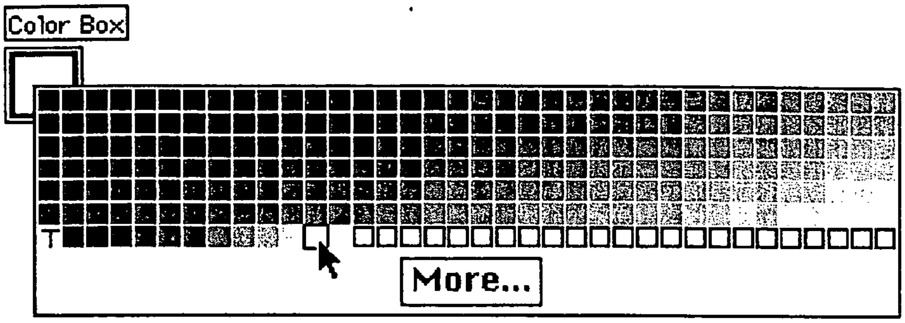
If you connect a numerical source to an enumeration indicator, it will convert to an enumerator item by pinning the value. If you connect an enumeration to a numerical data sink, the enumeration index is the number transmitted. If you wire an enumeration control to an enumeration indicator, you will get a bad wire if the enumeration items do not match.

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.



You can set the color of the color box by clicking on it with either the Operating or Coloring tool to display a color palette, as shown in the following illustration. Releasing your mouse button on the color you want selects that color.

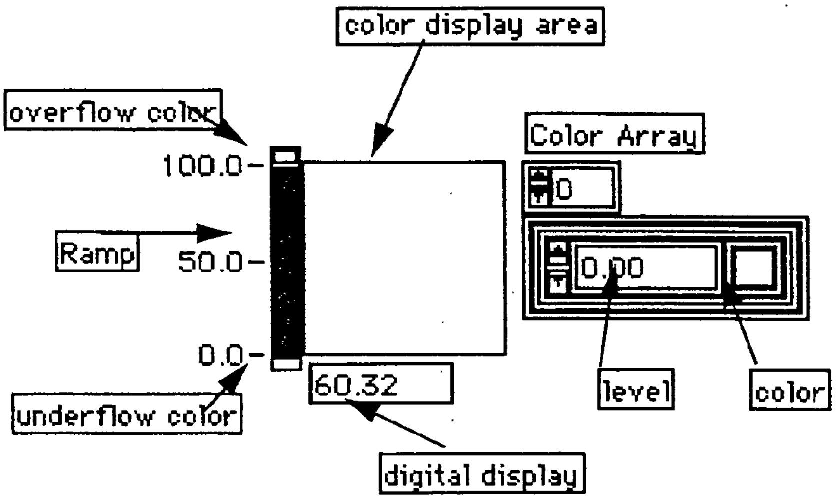


Color boxes are typically used on the front panel as indicators. The easiest way to set the color of a color box is to use the color box constant, from the **Structs & Constants** palette of the Functions menu in the Block Diagram Window. Wire the color box constant to the color box terminal on the block diagram. Clicking on the color box constant with either the Operating or Coloring tool displays the same color palette that you use to set the color box with. You can use a series of color box constants inside a case structure to change the color box indicator on the front panel to a variety of colors to indicate different conditions.

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 array, which consists of at least two pairs, 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.



You create these pairs using the color array. The color array index determines the number of levels. You select a level using the color array index, and then set the level, using the Operating Tool. Choose the color for each value by clicking in the color box with the Operating tool and selecting the color you want from the color palette that appears.

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 ramp has a number of options which you can display or hide. These options including the label, the color array, 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. These colors allow you 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.

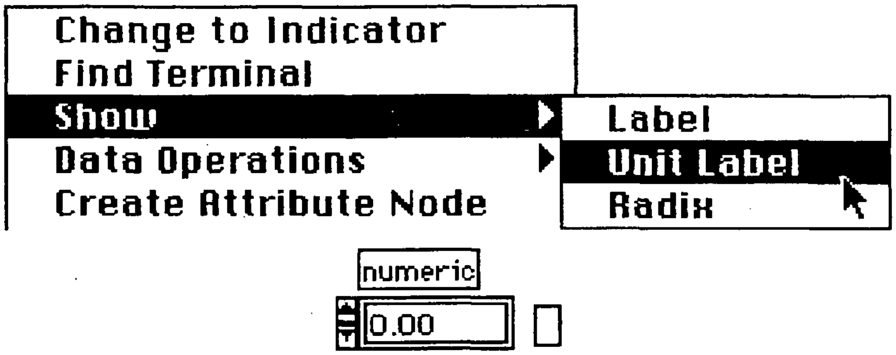
Both the color ramp and the color box are used to display a numeric value as a color. With the color ramp, you can assign any range of numbers to any range of colors. The color box, however, displays only the color specified by the red, green, and blue components of the numeric value; any given value always maps to the same color.

You can use the color ramp to specify color tables for the Intensity Graph and Intensity Chart controls. See Chapter 9, *Graph and Chart 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.

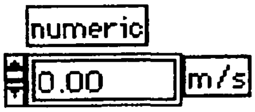
Units for a control are displayed and modified in a separate but attached label, called the unit label. You can show this label by selecting the option from the popup menu.



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.



The following tables list all the units you can use with the units feature

Table 5-2. Base Units

Quantity Name	Unit	Abbreviation
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

Table 5-3. Derived Units with Special Names

Quantity Name	Unit	Abbreviation
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

Table 5-3. Derived Units with Special Names (Continued)

Quantity Name	Unit	Abbreviation
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

Table 5-4. Additional Units in Use with SI Units

Quantity Name	Unit	Abbreviation
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

Table 5-5. CGS Units

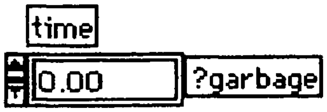
Quantity Name	Unit	Abbreviation
area	barn	b
force	dyne	dyn
energy	erg	erg
pressure	bar	bar

Table 5-6. Other Units

Quantity Name	Unit	Abbreviation
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

Entering Units

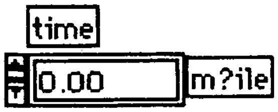
If you try to enter an invalid unit into a unit label, LabVIEW will flag the invalid unit by placing a ? in the label, as shown in the following illustration.



time

0.00 ?garbage

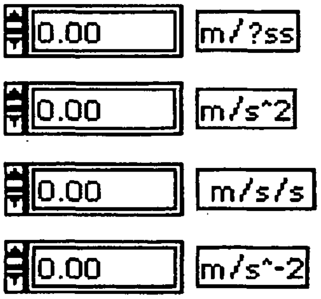
Notice that you must enter units using the correct abbreviations, as shown in the following illustration.



time

0.00 m?ile

Also, notice that LabVIEW will not allow you to enter ambiguous units. Thus m/ss is flagged, because it isn't clear whether you mean meters per second squared, or (meters/seconds) * seconds. To resolve ambiguity, you may have to enter the units differently, as shown in the following illustration.



invalid - 0.00 m/?ss

valid - 0.00 m/s^2

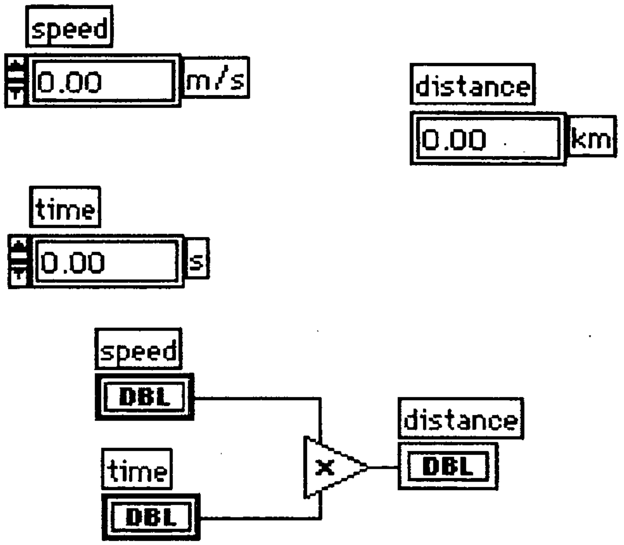
valid - 0.00 m/s/s

valid - 0.00 m/s^-2

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`.

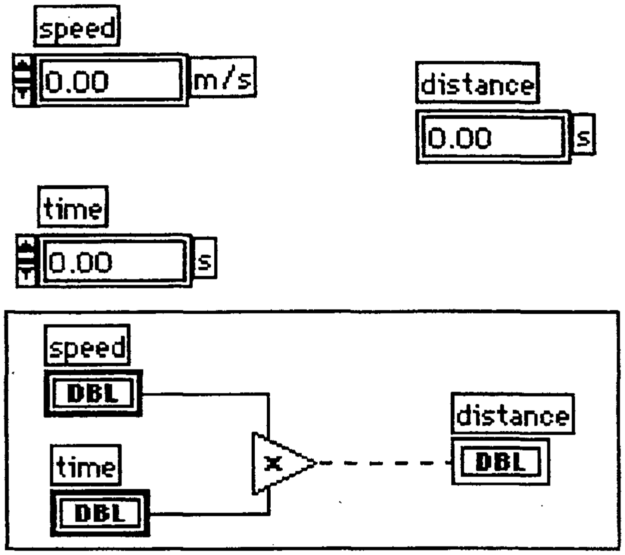
Units have Stricter Type Checking

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.



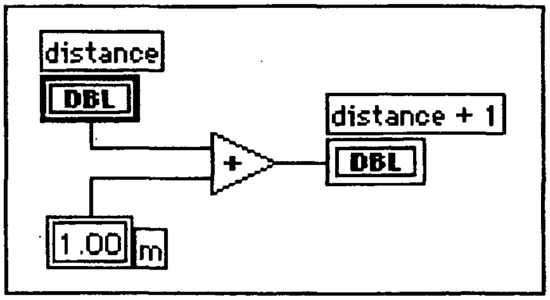
In the case of the previous illustration, the distance display will automatically be scaled to display kilometers instead of meters, since that was the specified unit for the indicator.

Notice that you cannot connect signals with incompatible units, as shown in the following illustration. If you select **List Errors** from the popup menu for the broken wire shown below, the error window indicates that the error is “Signal: 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.



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.

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 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 will show 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 will compute 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.

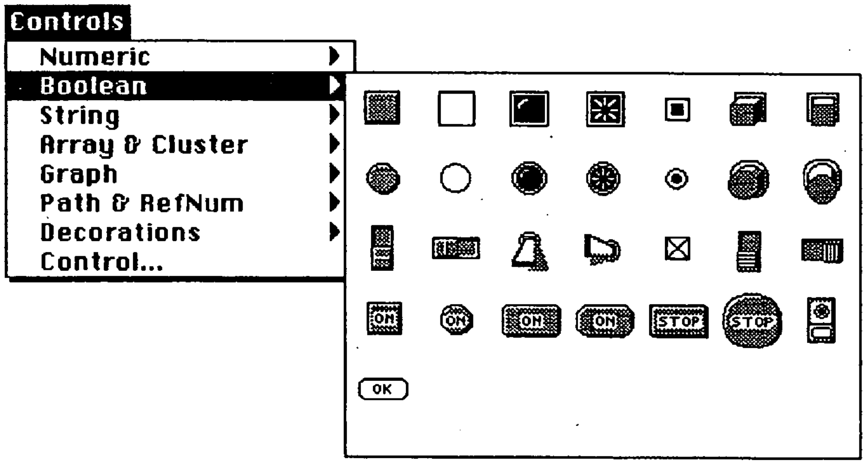
A polymorphic VI can have a polymorphic subVI because the respective units are kept distinct.

Chapter 6 Boolean Controls and Indicators

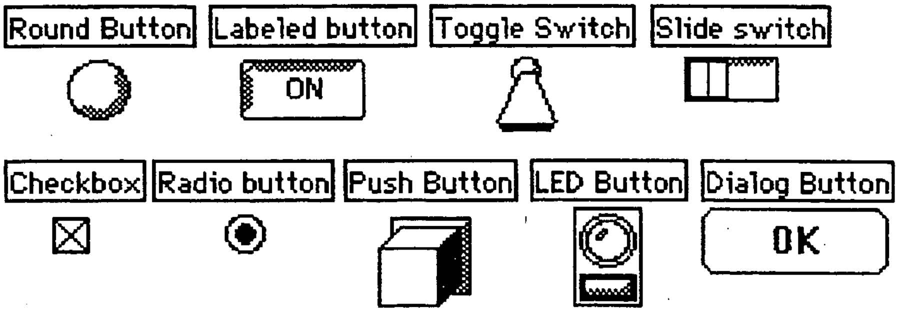
This chapter discusses how to create and operate Boolean controls and indicators.

Creating and Operating Boolean Controls and Indicators

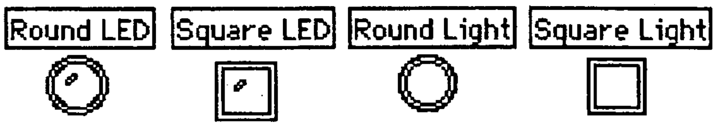
Boolean controls and indicators are available from the **Boolean** palette in the **Controls** menu.



Boolean controls that simulate mechanical-action push-button, toggle, and slide switches are shown below.



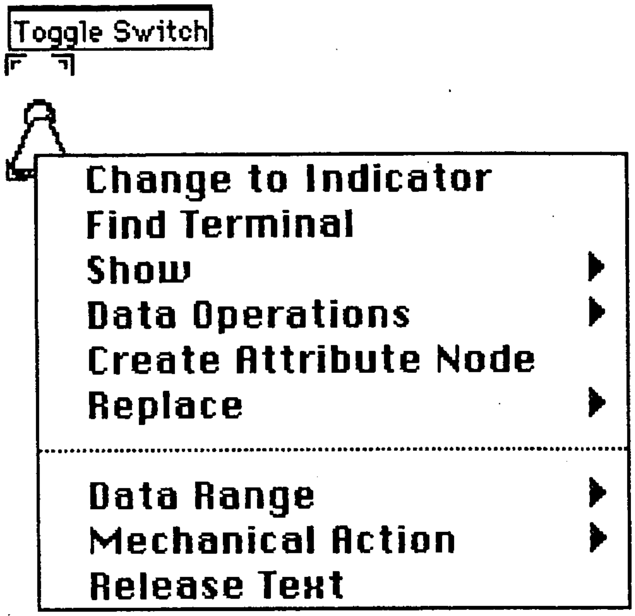
Boolean indicators that simulate LEDs are shown below.



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, while in edit mode, clicking toggles indicators off and on.

Configuring Boolean Controls and Indicators

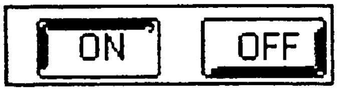
Each Boolean control or indicator has a default set of attributes. You can change these attributes through the object's pop-up menu. The pop-up menu for a Boolean object is shown below.



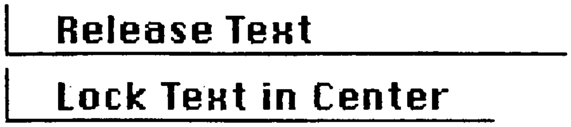
The options above the dotted line in the pop-up menu are common to all controls and indicators and are described in the Chapter 2, *Editing VIs*, of this manual. The options below the line are described in the following sections.

Labeling Booleans

Several controls in the **Boolean** palette are labeled Booleans. Initially, the buttons display the word ON in their TRUE state and the word OFF in their FALSE state. (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. Notice that **Release Text** is then replaced in the pop-up menu by **Lock Text in Center.**



Now you can use the Positioning tool to reposition the text, or choose the option **Lock Text in Center**. Use the **Text** menu to change the font, size and color of the Boolean text. Selecting **Use Default Font** changes the Boolean text to the default font, which you can set in the **Text Style...** dialog box. You can also remove the button label from either state by selecting **Hide Boolean Text** from the button pop-up menu.

The Boolean objects 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.

If you want to change the font of either the 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

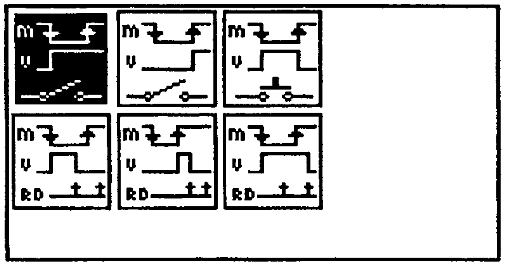
If you want to suspend a VI if a conditional test fails to produce the correct Boolean value, select **Suspend if True** or **Suspend if False** from the **Data Range** submenu of the Boolean pop-up menu.

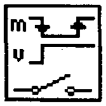
Boolean Range Checking

You may want to detect errors in Boolean values. If you expect a Boolean to be TRUE always, you can select **Suspend If False** from the **Data Range** submenu of the Boolean pop-up menu. If you expect the Boolean to be FALSE always, 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.

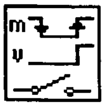
Configuring the Mechanical Action of Boolean Controls

Boolean controls have six types of mechanical action. Select the appropriate action for your application from the pop-up menu **Mechanical Action** palette. In these palette symbols, M stands for the motion of the mouse button when you operate the control, V stands for the control's output value, and RD stands for the point in time that the VI reads the control.

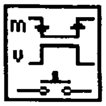




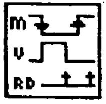
The **Switch When Pressed** action changes the control value each time you click on it with the operating tool, in a manner similar to that of a light switch. The action is not affected by how often the VI reads the control



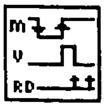
The **Switch When Released** action changes the control value only after you release the mouse button during a mouse click within the control's graphical boundary. The action is not affected by how often the VI reads the control.



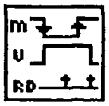
The **Switch Until Released** action changes the control value when you click on it, and retains the new value until you release the mouse button, at which time the control reverts to its original value, similar to the operation of a door buzzer. The action is not affected by how often the VI reads the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

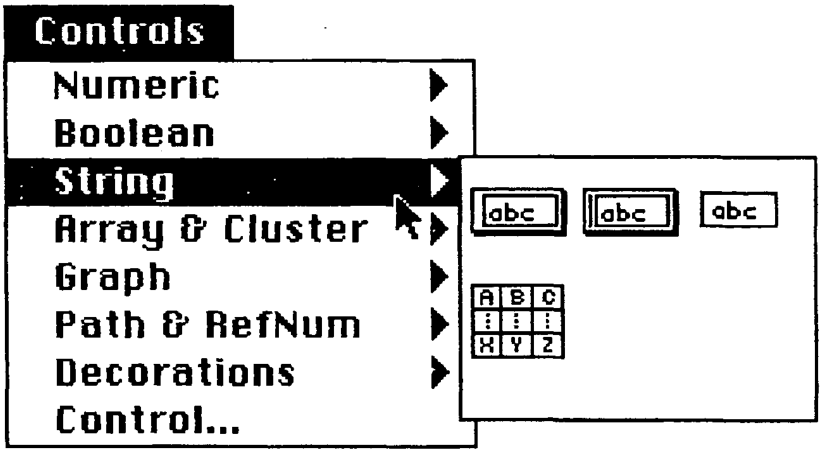
Customizing a Boolean with Imported Pictures

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 the *Creating a Custom Control* section of Chapter 4, *Introduction to Front Panel Objects*.

Chapter 7

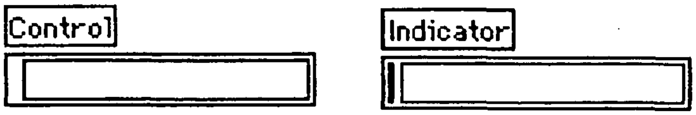
String and Table Controls and Indicators

This chapter discusses how to use string controls and indicators, and the table. You can access these objects through the **String** palette of the **Controls** menu.



Using String Controls and Indicators

A string control and string indicator are shown in the following illustration.



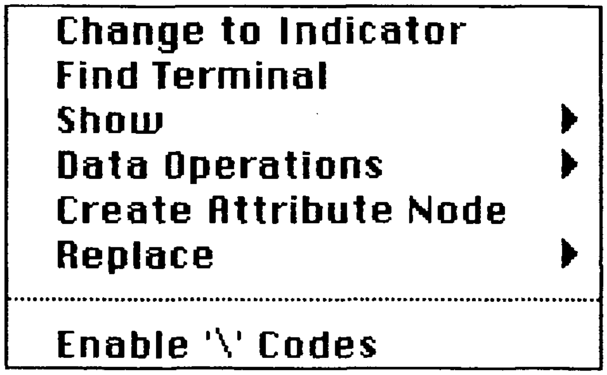
You enter or change text in the string control using the Operating tool or the Labeling tool. As with the digital control, new or changed text does not pass to the diagram until you press the enter key on the numeric keypad, click on the enter button in the tool palette, or click outside the control to terminate the edit session. Pressing the <Enter> 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: ***In run mode, you use the tab key to move to the next control. In edit mode, pressing the tab key changes tools. To enter a tab character into a string, select the Enable '\' Codes option from the string pop-up menu and type*** `\t`***. To enter a carriage return into a string, press the return key on the alphanumeric keyboard, or select the Enable '\' Codes option from the string pop-up menu and type*** `\r`***. To enter a line feed into a string, type*** `\n`***. See Table 7-1, LabVIEW '\' Codes, for a complete list of '\' codes.***

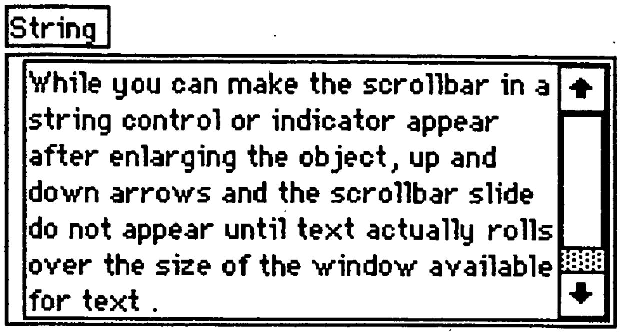
Options for String Controls and Indicators

Strings have special features that you can access through the string pop-up menu, shown below.



Using the Scrollbar with String Controls and Indicators

If you choose the **Scrollbar** option from the string pop-up **Show** submenu, a vertical scrollbar appears on the string control or indicator as shown below 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.



While you can make the scrollbar in a string control or indicator appear after enlarging the object, up and down arrows and the scrollbar slide do not appear until text actually rolls over the size of the window available for text .

Entering and Viewing Nondisplayable Characters

Choosing the option **Enable '\' Codes** 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.

Table 7-1. LabVIEW '\' Codes

Code	LabVIEW Interpretation
\00 - \FF	Hex value of an 8-bit character; must be uppercase
\b	Backspace (ASCII BS, equivalent to \08)
\f	Formfeed (ASCII FF, equivalent to \0C)
\n	New line (ASCII LF, equivalent to \0A)
\r	Return (ASCII CR, equivalent to \0D)
\t	Tab (ASCII HT, equivalent to \09)
\s	Space (equivalent, \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 LabVIEW interprets `\bFare` and `\bfare` as a backspace followed by the words *Fare* and *fare*. In the sequence `\Bfare`,\B is not the backspace code, and \Bf is not a valid hex code. In a case like this, when a backslash is followed by only part of a valid hex character, LabVIEW assumes a 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 **Enable '\' Codes**. 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 `\r` `\\`.

```
left\n\\right\\3F
```

Suppose now that you select **Enable '\' Codes** 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
ight?
```

Now if you select **Enable '\' Codes** again, the following string appears.

```
left\n\right?
```

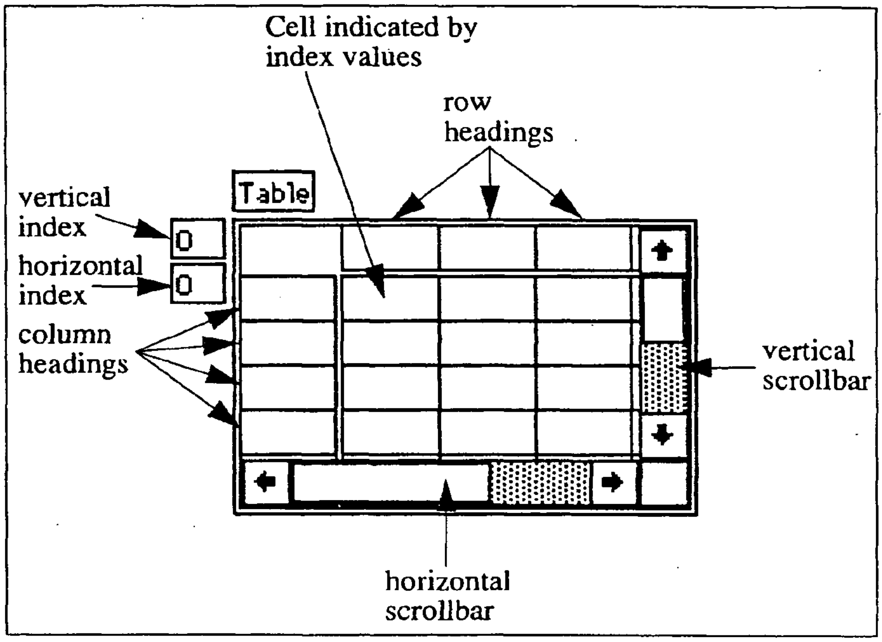
Indicators behave the same way.

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.

Using the Table

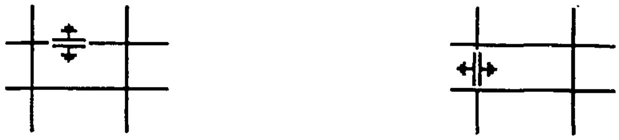
A table is a 2D array of strings. The following illustration is an example of a table with all its features shown.



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.

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 in the following illustration appears. Click and drag to resize the row or column.

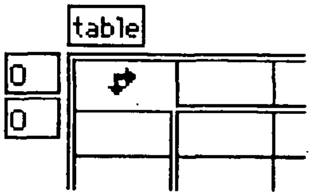


You can use the keyboard to enter data into a table rapidly. Click inside a cell with either the Operating tool or the Labeling tool, and enter your data from the keyboard.

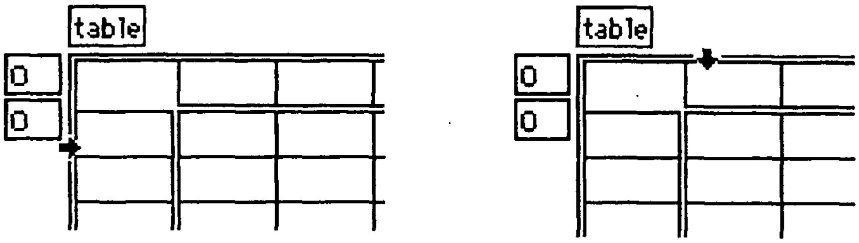
When you are in run mode or edit mode, the <Enter> key on the alphanumeric keyboard will enter your text and move the cursor into the cell below. The <Enter> key on the numeric keypad will enter your data and terminate entry. Pressing the <Shift> or the <Ctrl> key while pressing the arrow keys moves the entry cursor through the cells.

You can also 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. Select data using the Operating or Labeling tool. You can double-click in a cell to select it, or click in a cell and drag to select a number of cells for copying or pasting. Moving outside the current contents of the table will scroll the table while extending the selection. A border appears around selected cells to indicate that they have been selected.

You can also select all the data in a table, or all the data in a row or column. Position the Operating or Labeling 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. Click the mouse to select the data.



To select an entire row or column of data, position the Operating tool or Labeling 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. Click and drag across width of column or height of row to select the data.



If you cut the top row of data, all rows below it scroll up. If you cut the left-most column of data, all columns scroll left.

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.

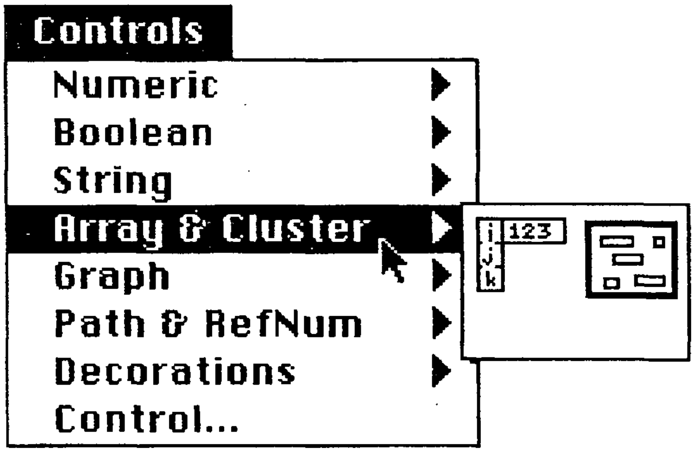
The table is the same as a two-dimensional array of strings as far as the LabVIEW block diagram is concerned. Because of this, LabVIEW functions that operate on a 2D array of strings (such as the Array To Spreadsheet String function, for example) can be particularly helpful in using the table. See Chapter 6, *String Functions*, of the *LabVIEW Function Reference Manual* for information on using the these functions.

Chapter 8

Array and Cluster Controls and Indicators

This chapter describes how to use LabVIEW arrays and clusters. The *LabVIEW Function Reference Manual* describes LabVIEW functions, many of which can operate on arrays in addition to scalars. Chapter 7, *Array and Cluster Functions*, of the *LabVIEW Function Reference Manual* describes the functions designed exclusively for array and cluster operations.

You access arrays and clusters from the **Array & Cluster** palette of the **Controls** menu.



Arrays

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.

index	0	1	2	3	4	5
11-element array	Melissa	Greg	Gregg	Don	Duncan	Thad

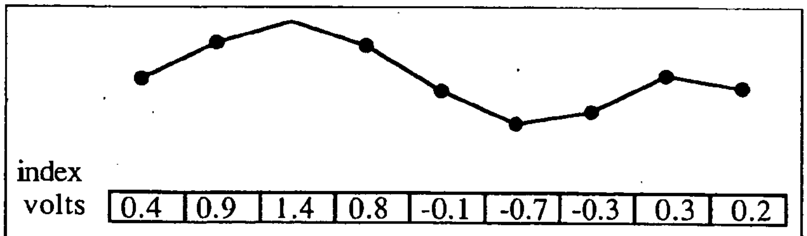
index (continued)	6	7	8	9	10
11-element array (continued)	Dean	Stepan	Kate	Mary	Mark

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.

Jeff	Paul	Rob	Bruce	Steve	Meg	Jack	Brian	Deb	Kevin	Tom

List of Names in an Array of Strings

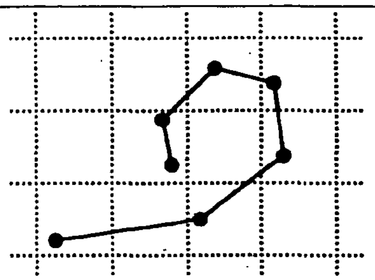
Another example is a waveform represented as a numeric array in which each successive element is the voltage value at successive time intervals, as shown in the following illustration.



index									
volts	0.4	0.9	1.4	0.8	-0.1	-0.7	-0.3	0.3	0.2

Waveform in an Array of Numbers

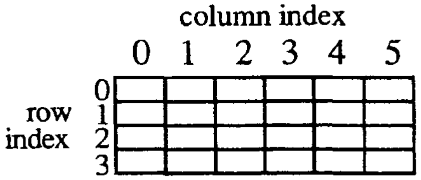
A more complex example is a graph represented as an array of points where each point is a pair of numbers giving the X and Y coordinates, as shown in the following illustration.



index							
X coord	0.4	2.2	3.3	3.2	2.4	1.8	1.9
Y coord	0.2	0.5	1.3	2.3	2.6	1.9	1.2

Graph in an Array of Points

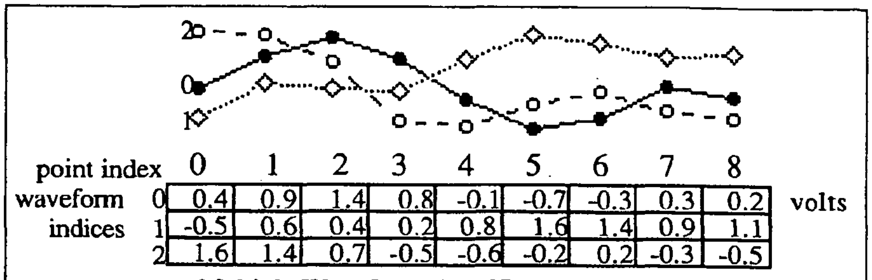
The preceding examples are *one-dimensional* arrays. A *two-dimensional* array requires two indices to locate an element: a column index and a row index, both of which are zero-based. In this case we speak of an N column by M row array, which contains N times M elements, as shown in the following illustration.



6 column by 4 row array of 24 elements

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. Below is a collection of waveforms represented as a two-dimensional array of numbers. The row index selects the waveform and the column index selects the point on the waveform.



point index	0	1	2	3	4	5	6	7	8	
waveform 0	0.4	0.9	1.4	0.8	-0.1	-0.7	-0.3	0.3	0.2	volts
indices 1	-0.5	0.6	0.4	0.2	0.8	1.6	1.4	0.9	1.1	
2	1.6	1.4	0.7	-0.5	-0.6	-0.2	0.2	-0.3	-0.5	

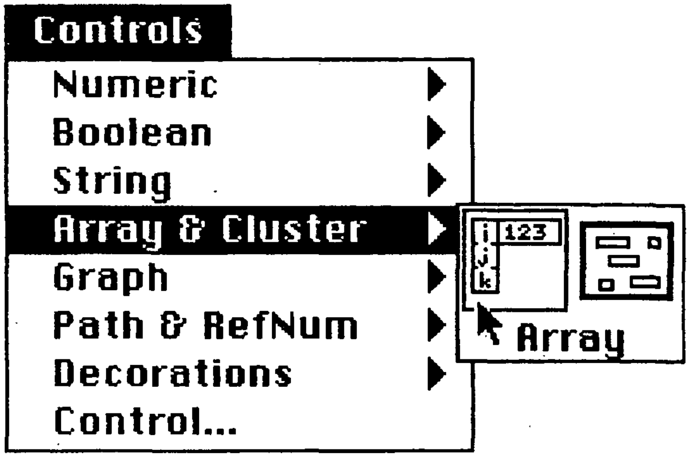
Multiple Waveforms in a 2D Array of Numbers

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.

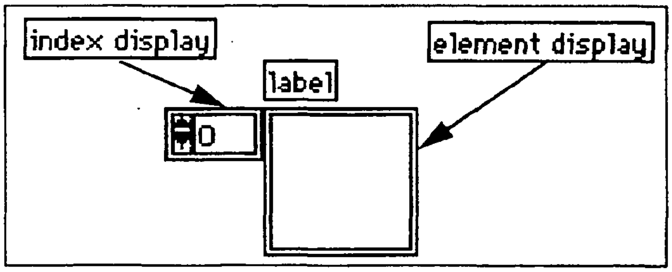
Creating Array Controls

You create an array control in LabVIEW by first selecting an array from the **Array & Cluster** palette of the **Controls** menu.



You create a LabVIEW array control or indicator by combining an *array shell* from the **Array & Cluster** palette of the **Controls** menu, as shown above, 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.

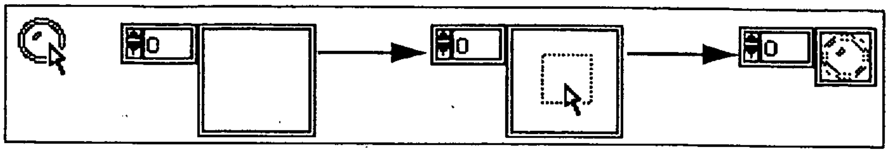
Selecting **Array** from the **Array & Cluster** palette places an array shell on the front panel, as shown in the following illustration.



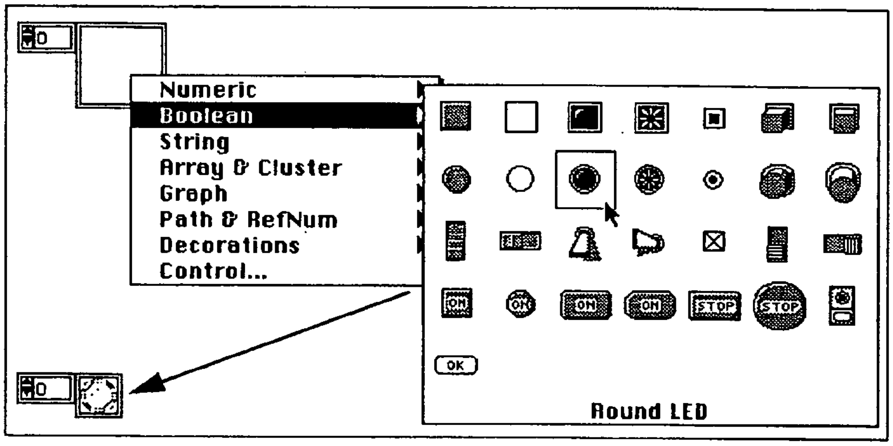
A new array shell has one *index display*, an empty *element display*, and an optional label.

There are two ways for defining the type of the array. 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.

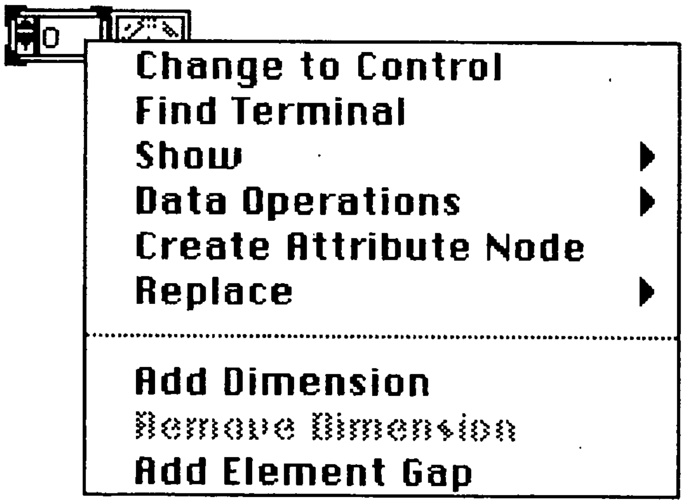
For example, if you want to define an array of Booleans, you can either drag a Boolean into the element display area, or you can pop up in the empty element area and select a Boolean control. Both methods are shown in the following illustration.



or

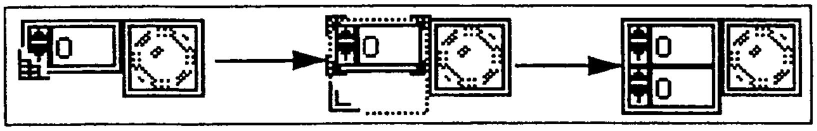


The pop-up menu of the index display is shown in the following illustration.

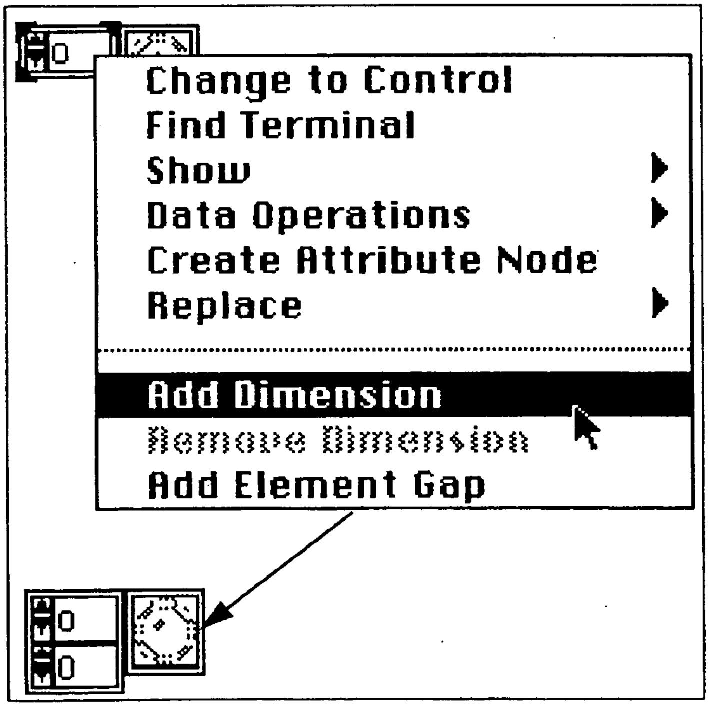


Setting the Array Dimension

A new array has one dimension and one index display. You resize the index vertically or select the **Add Dimension** command from the array shell pop-up menu to add a dimension, and 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.



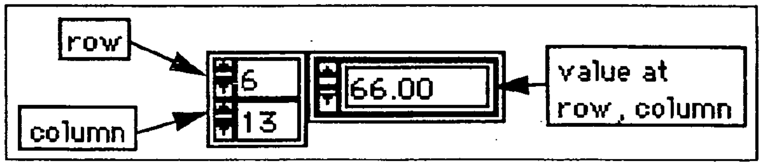
or



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

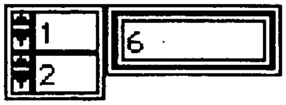
If you think of a two-dimensional array as rows of columns, the top display is the row index and the bottom display is the column index. The combined display at the right shows the value at the specified position, as shown in the illustration below.



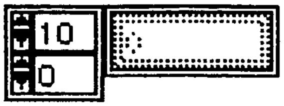
For example, let's say you have an array of 3 rows and 4 columns, with values as shown in the following table.

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, etc.,. 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 that 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 above, then the display would be grayed out as shown in the following illustration.

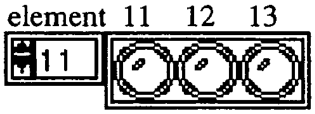


Displaying an Array in Single-Element or Tabular Form

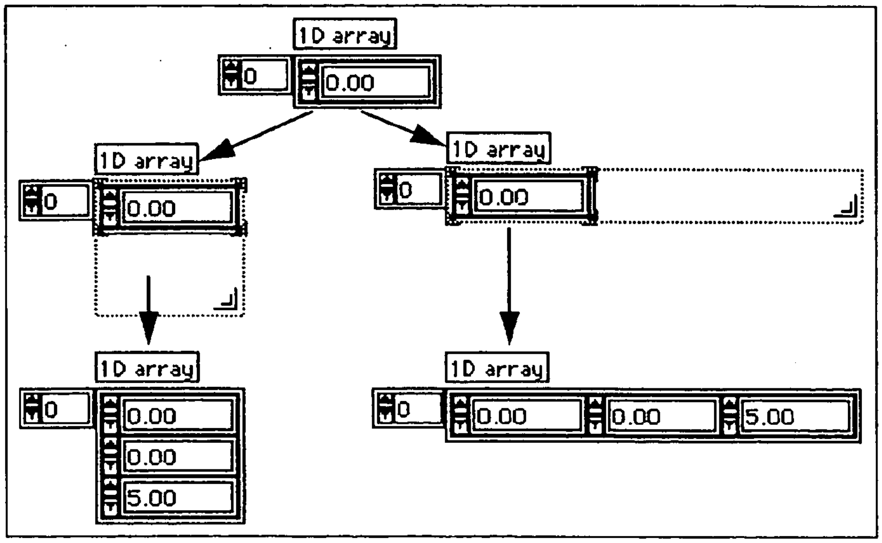
array Resizing tool

usual Resizing symbol

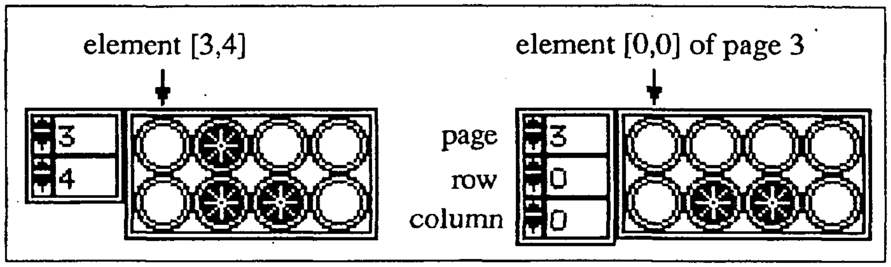
A new array appears in single-element form. The array displays the value of a single element–the one referenced by the index display. You can also display the array as a table of elements by resizing the array shell from any of the four corners surrounding the element. The array Resizing tool is slightly 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.



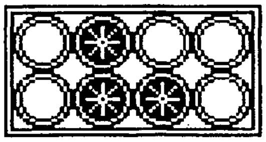
For example, the following illustration shows how you could resize a one-dimensional array either vertically or horizontally to show more elements simultaneously.



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 leftmost 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, turn off the **Index Display** option from the **Show** menu 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 discussed in the *LabVIEW Function Reference Manual*.

Operating Arrays

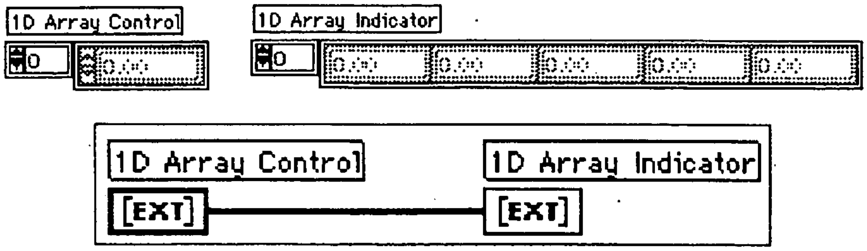
You operate the array's element exactly as you would if it were not part of the array. You operate the index display the same way you operate a digital control. To set the value of an array control element, bring the element into view with the index display, then set the element's value.

Setting the Default Size and Values of an Array

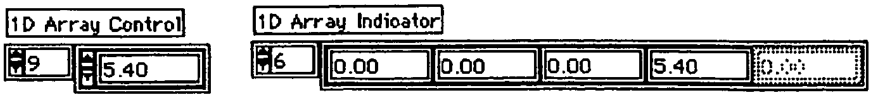
You cannot fix the size of a LabVIEW array. However, when you set the default values of the array, you also set the default size. *Do not make the default size larger than necessary.* Keep in mind that if you set an array to have a large default value, all of the default data is saved along with the 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).

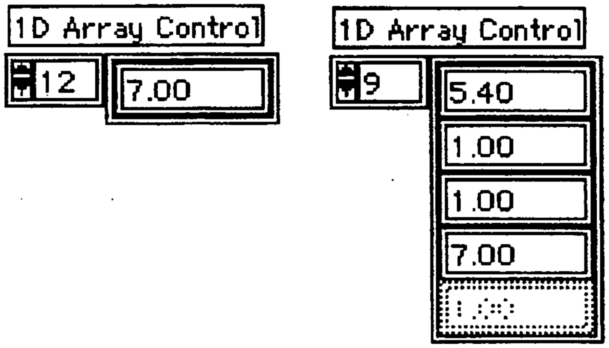


If you set the index display of the array control to 9 and set the element value at index 9 to 5.4, the data in the array expands to 10 elements (0 to 9). The value of element 9 is the value you set, 5.4 in this example. LabVIEW gives the other elements the default value of the digital control, 0.00. 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 choosing the **Data Range...** option from 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.



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 $[N_i$ by N_j by $N_k...]$ to $[M_i$ by M_j by $M_k...]$, assign a value to the new last element $[M_{i-1}, M_{j-1}, M_{k-1}...]$. To decrease the size, first execute the **Empty Array** command from the array shell **Data Operations** 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 **Cut Data** option from the **Data Operations** pop-up menu, as explained in the following sections.

Finding the Size of an Array

To find the size of an array control or indicator, select **Show Last Element** from the **Data Operations** submenu in the array shell 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 will separate 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 Panel window border before releasing the mouse button. You also cancel a resizing operation by dragging the border past the Panel window before releasing the mouse button. If the array is part of the current selection, then grabbing an element will drag the array.

You can resize the array index vertically from any corner, or horizontally from the left.

Selecting Array Cells

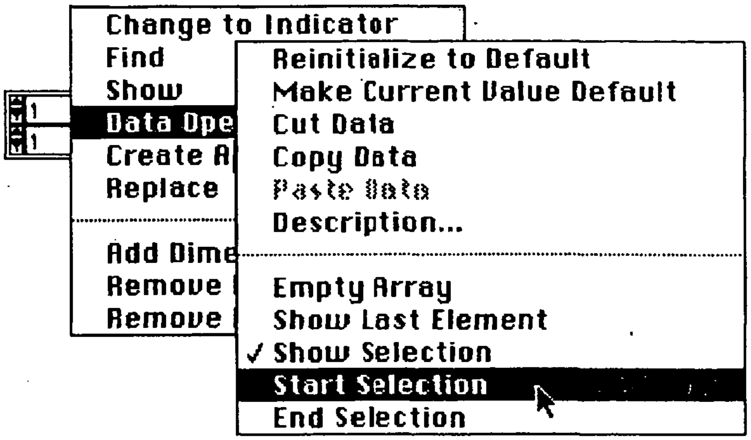
You can copy data from and paste data to LabVIEW arrays. To select the data, set the array index to the first element in the data set you want to copy and select the **Start Selection** option from the **Data Operations** submenu of the array pop-up menu. Set the array index to the last element in the data set you want to copy and select the **End Selection** option from the **Data Operations** submenu of 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.

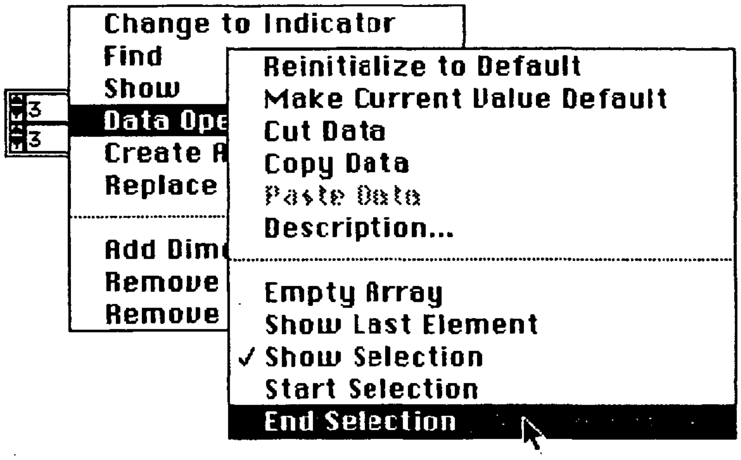
Begin by selecting **Show Selection** from the pop-up menu. Show Selection must be activated for your selected cells to be visibly marked.

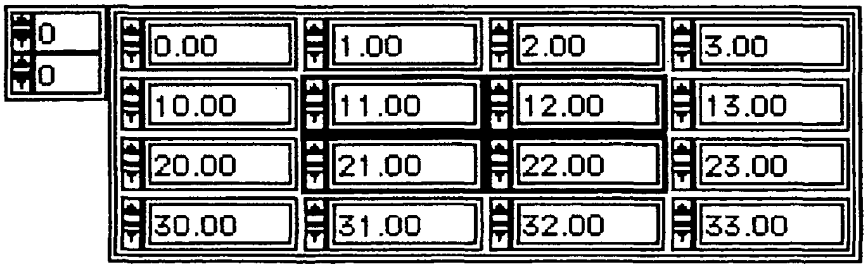
A border surrounds selected cells. You can select **Add Element Gap** from the pop-up menu. This separates the cells by a thin open space. While it is not necessary to add the element gap, this open border makes the array better looking. When you select cells, this open space is filled by a thick border, 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 will be surrounded by a blue border (thick black on a monochrome system).







0.00	1.00	2.00	3.00
10.00	11.00	12.00	13.00
20.00	21.00	22.00	23.00
30.00	31.00	32.00	33.00

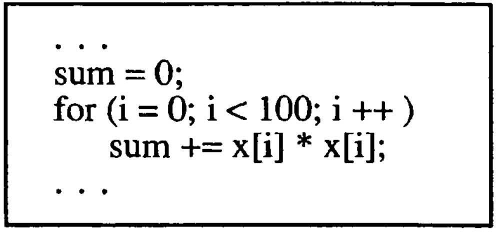
Notice that the selection includes the cells denoted by lower index numbers but not the cells denoted by the higher index numbers. If you want to include the cells from `1,1` to `3,3`, you would have set the index to `4,4` before choosing **End Selection**.

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 will remain 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

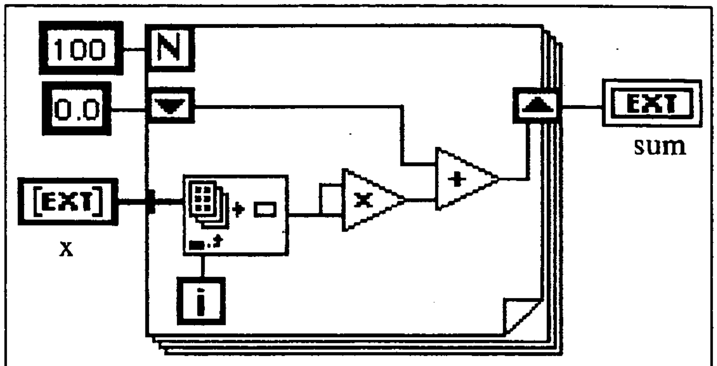
Most programming languages have arrays, although few have the number of built-in array functions that LabVIEW has. Other languages typically stop at the fundamental array operations of extracting an element or replacing an element, leaving it to the users to build more complex operations themselves. 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 LabVIEW's higher-level array functions. The example below 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.

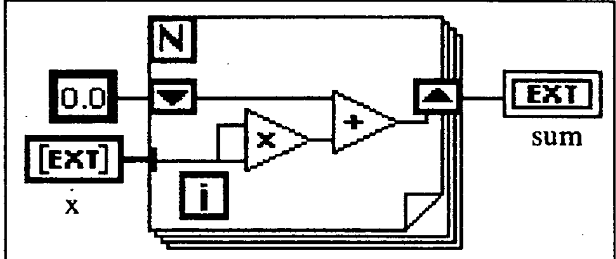


```
...
sum = 0;
for (i = 0; i < 100; i ++ )
    sum += x[i] * x[i];
...
```

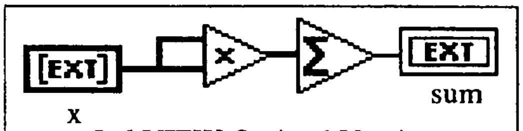
C Code Fragment



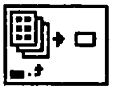
LabVIEW Direct Translation



LabVIEW Auto-Indexing Version



LabVIEW Optimal Version

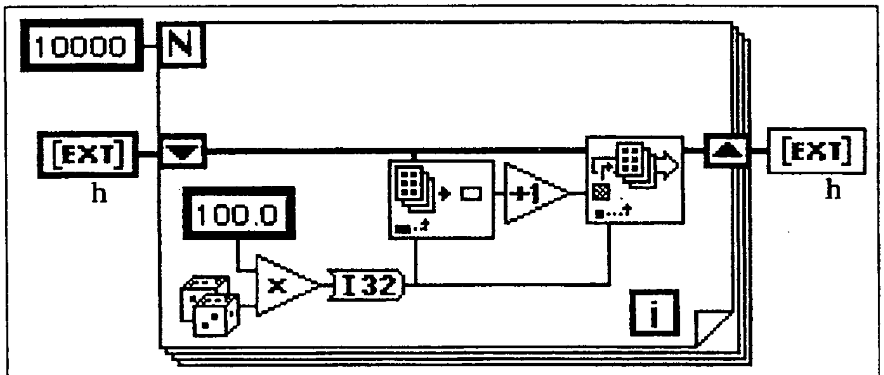


Index Array function icon

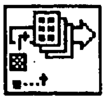
The indexing operation for extracting an element is represented in C by brackets [ ] following the array name and enclosing the index. In LabVIEW, this operation is represented by the Index Array function icon shown at left. The array is wired to the top left terminal, the index is wired to the bottom 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.

```
    . . .
    DO 10 I = 1, 10000
    J = 100.0 * RAND( ) + 1
    H(J) = H(J) + 1
 10 CONTINUE
    . . .
```



LabVIEW Direct Translation



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

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.

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.

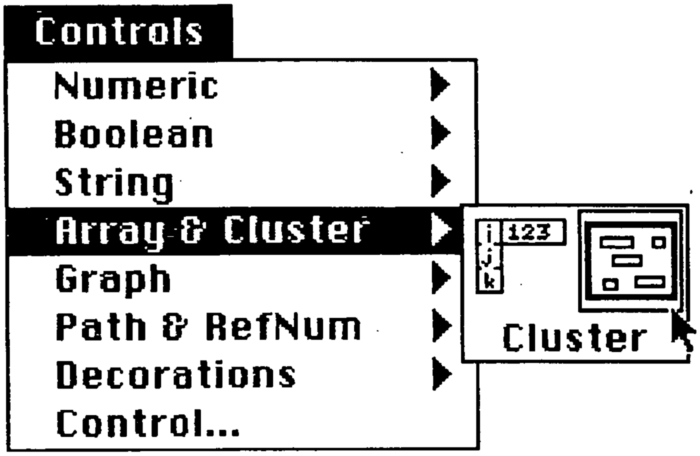
▨ Common Cluster Wire Pattern

▨ 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.

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** menu, as shown below.



A new cluster shell has a resizable border and an optional label.

When you pop-up in the empty element area, the **Controls** menu appears. You can place any element from the **Controls** menu 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

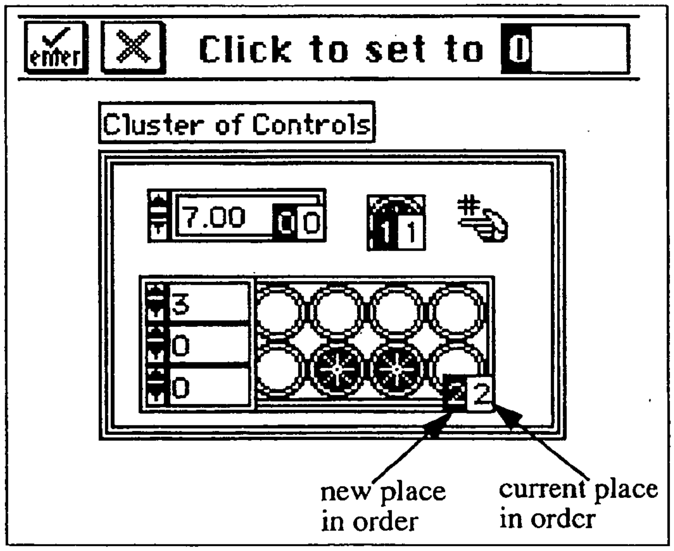
With the exception of setting default values and the **Change to Control** and **Change To Indicator** options, you configure controls and indicators the same way you configure controls and indicators that are not in a cluster.

Setting Cluster Default Values

Selecting the **Make Current Value Default** command from the **Data Operations** menu of a cluster pop-up menu sets the default value of all individual cluster elements to the current value of each individual element. In the same way, selecting the **Reinitialize To Default Values** command from the **Data Operations** menu of a cluster element pop-up menu resets all elements to their individual default configuration. To change the default value of a single element, use the **Data Range...** option from the element's pop-up menu, set the default before you drag the object into the cluster, or pop up on the individual element and select **Make Current Value Default**, after setting the proper value with the operating tool.

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 figure.



The white boxes on the elements show their current places in the cluster order. The black boxes show the element's new place in the order. Clicking on an element with the cluster order cursor sets the element's

☑ enter

enter button

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.

☒

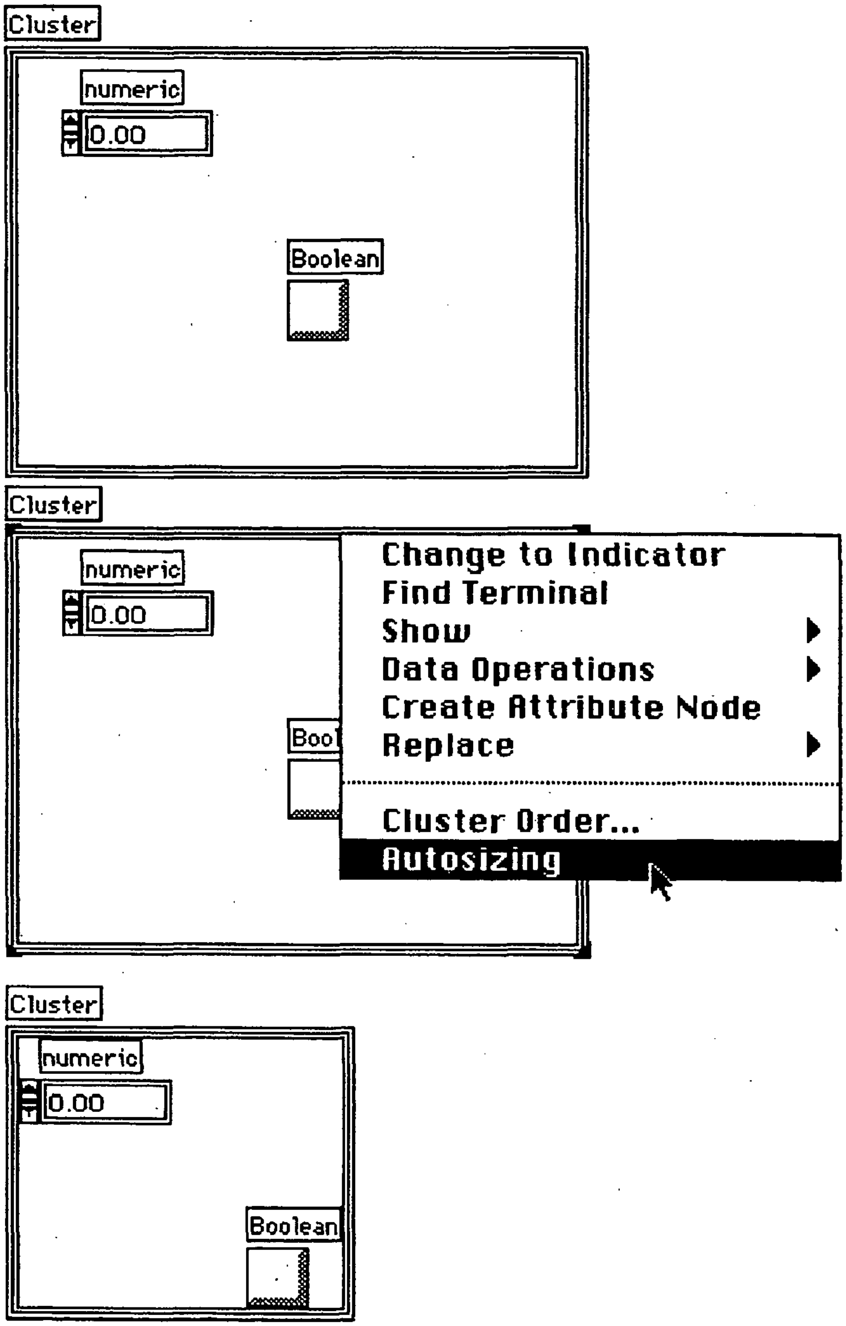
X button

The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions in the block diagram. See the discussion of the Bundle and Unbundle functions in the *LabVIEW Function Reference Manual* for more information.

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 Panel window border before releasing the mouse button. You also cancel a resizing operation by dragging the border past the Panel window before releasing the mouse button.

You can shrink clusters to fit their contents by selecting Autosizing from the pop-up menu.



Assembling Clusters

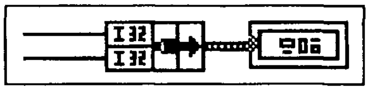
LabVIEW has three functions for assembling or building clusters. The Bundle and Bundle By Name functions assembles a cluster from individual elements or replaces individual elements with elements of the same type. The Array To Cluster function converts an array of elements into a cluster of elements.

Bundle Function

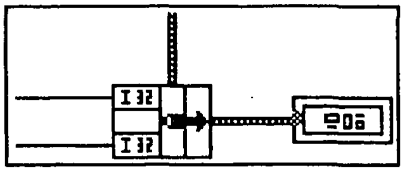
Bundle function

The Bundle function, which you obtain from the **Array & Cluster** palette of the **Functions** menu, 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. 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. You use this terminal to replace one or more elements of an existing cluster wire without affecting the other elements.



See the *Replacing Cluster Elements* section in this chapter for an example of using this function to replace elements in a cluster.

Bundle By Name Function

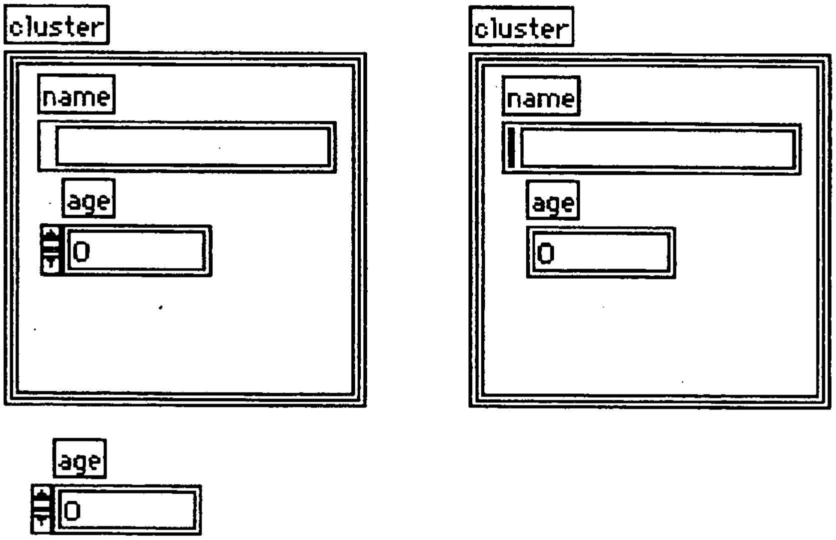
Bundle By Name function

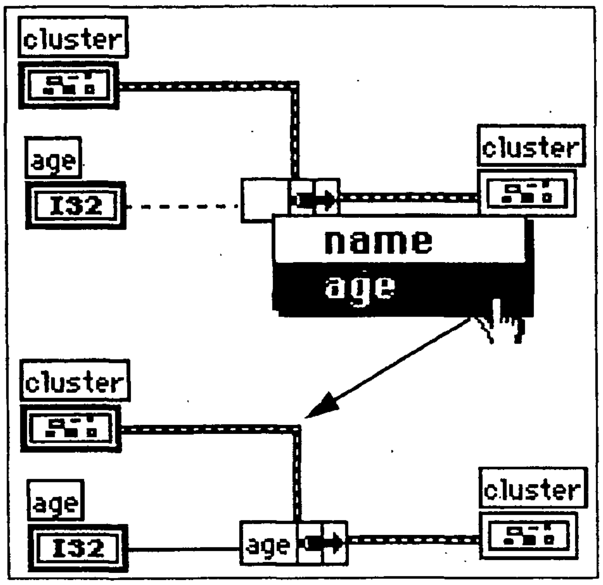
The Bundle By Name function, also in the **Array & Cluster** palette of the **Functions** menu, is similar to the Bundle function. Instead of referencing fields by position, however, you reference them by name. Unlike the Bundle function, you can access only the elements that you want to access. For 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.

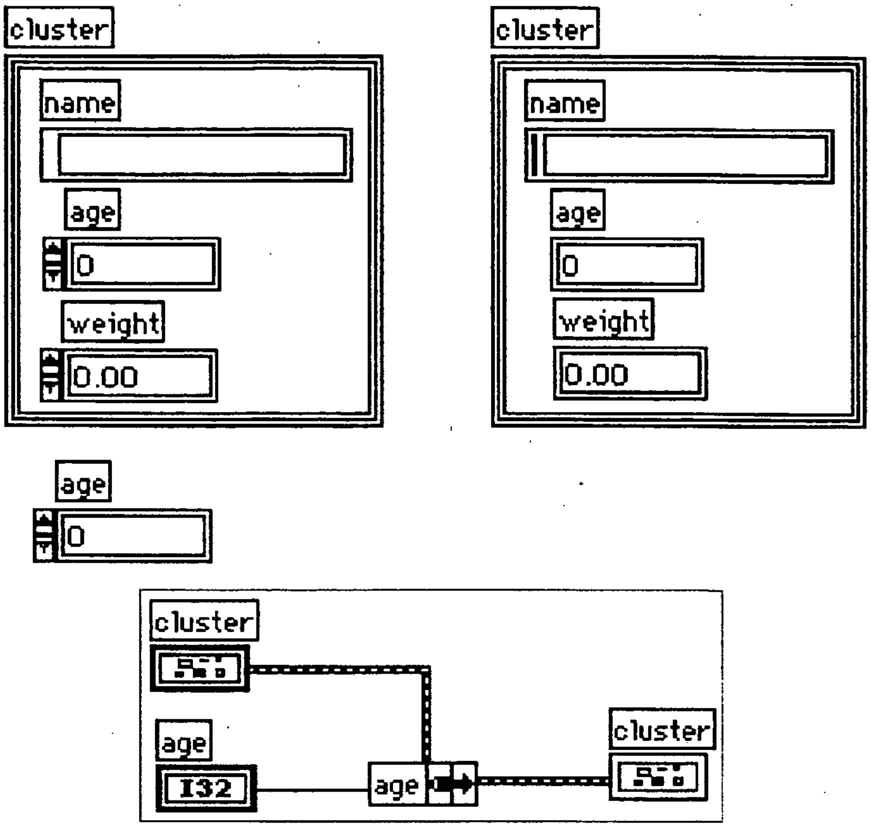




Notice that you can resize the Bundle By Name function to show as many or as few elements as you need. This is different from the Bundle function, which requires you to size it to have the same number of elements as are in the resulting cluster.

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 will 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, weight, to the source and destination clusters, the VI is still correct.



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 definition's 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.

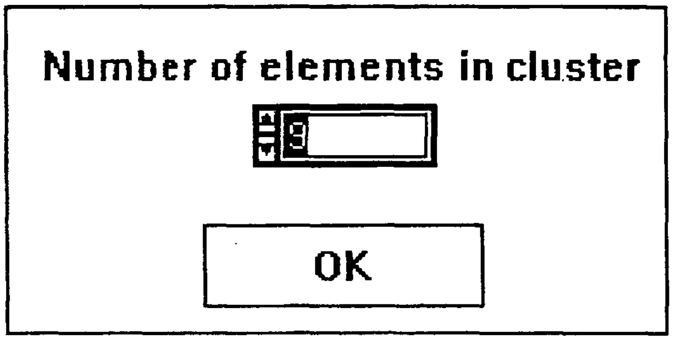
See the *Type Definitions* section of Chapter 4, *Introduction to Front Panel Objects*, for more information on type definitions. Also see the *Unbundle By Name Function* section of this chapter.

Array To Cluster Function

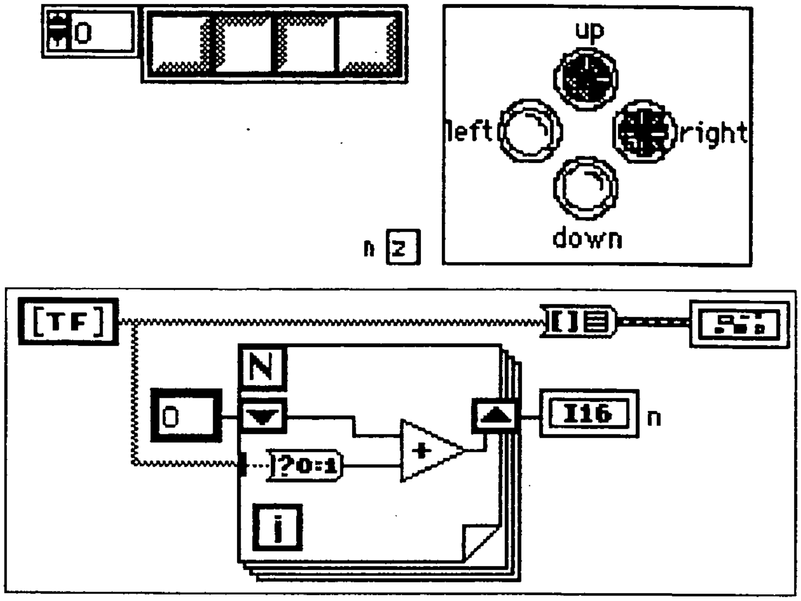
Array To Cluster function

The Array To Cluster function, which you select from the **Conversion** palette of the **Functions** menu, 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 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 below 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 format–horizontal with the lowest element at left and an attached index display.



Disassembling Cluster Elements

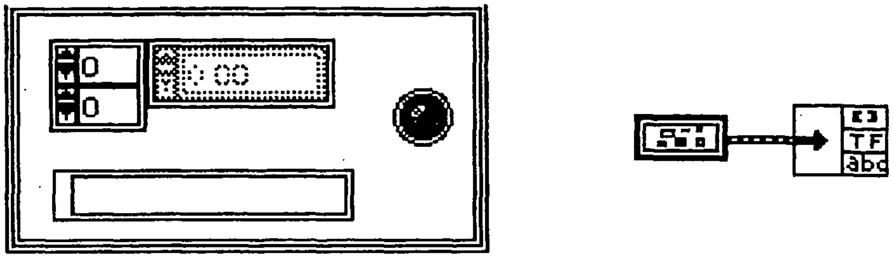
LabVIEW has three functions for disassembling clusters. The Unbundle and Unbundle By Name functions disassemble a cluster into individual elements, and the Cluster To Array function converts a cluster of identically typed elements into an array of elements of that type.

Unbundle Function

Unbundle function

The Unbundle function, which you obtain from the **Array & Cluster** palette of the **Functions** menu, 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 below.

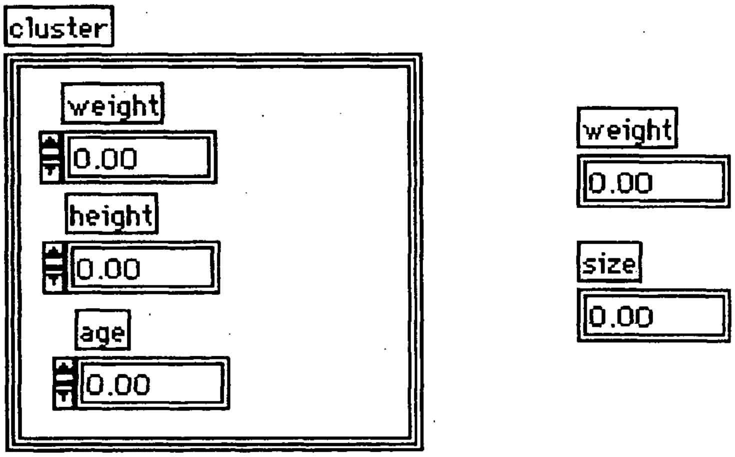


Unbundle By Name Function

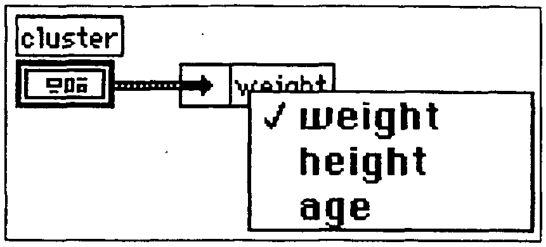
Unbundle By Name function

The Unbundle By Name function, also in the **Array & Cluster** palette of the **Functions** menu, 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 don't have to have one output for every cluster element.

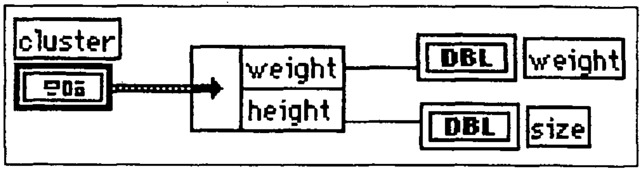
For example following is a panel with a cluster of three elements, weight, height and age.



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.

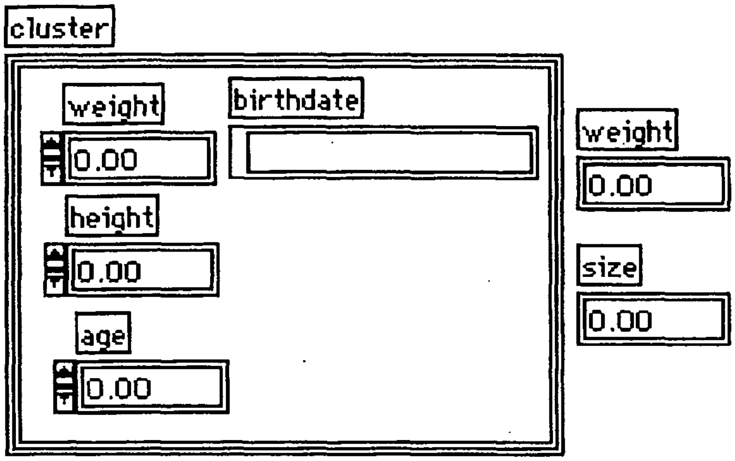


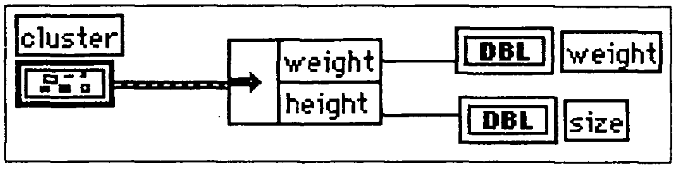
You can resize the function to read other elements. Note that you do not have to read all of the elements, and that you can read them in arbitrary order.



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 always has to have exactly the same number of terminals as there are 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.





As with Bundle By Name, Unbundle By Name is particularly useful in larger applications in conjunction with type definitions. See the *Bundle By Name Function* section of this chapter and the *Type Definitions* section of Chapter 4, *Introduction to Front Panel Objects*, for more information.

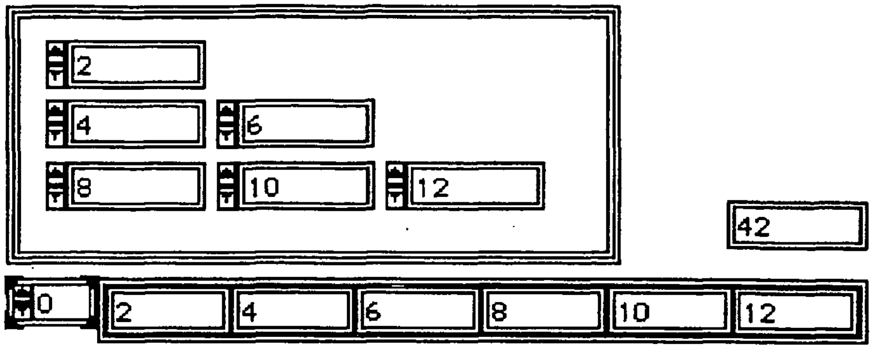
Cluster To Array Function

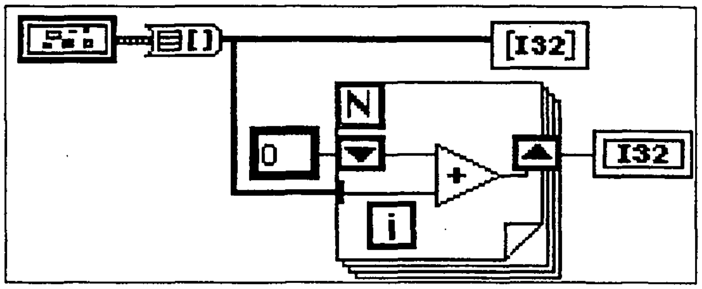
Cluster To Array function

The Cluster To Array function, which you obtain from the **Conversion** palette of the **Functions** menu, 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.

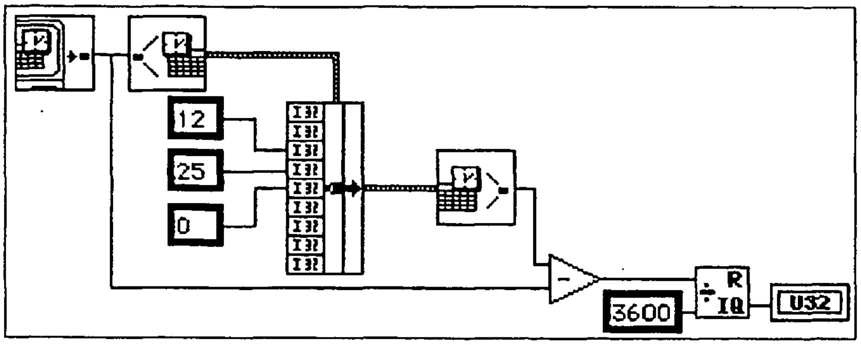
The Cluster To Array function is useful when you want to group a set of front panel controls in a particular order within a cluster but you want to process them as an array on the block diagram, as shown in the following illustration.





Replacing Cluster Elements

Sometimes you want to replace 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 Christmas 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.



Chapter 9

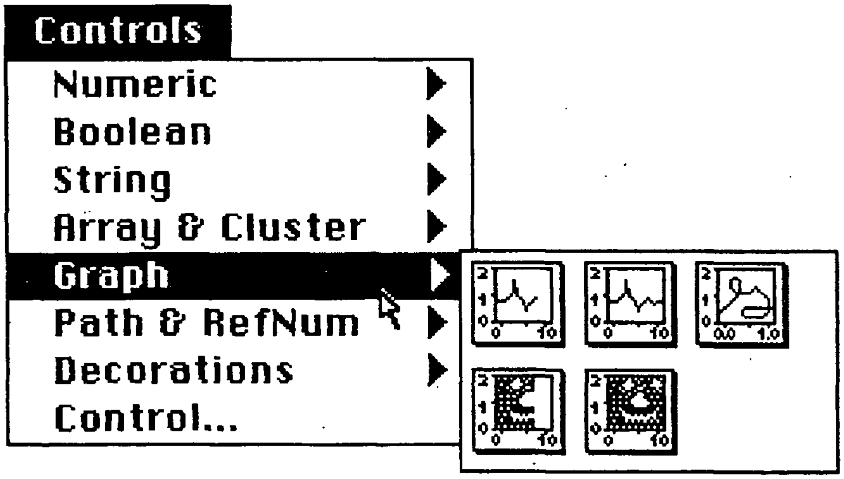
Graph and Chart Indicators

This chapter describes how to create and use the graph and chart indicators in the **Graphs** palette of the **Controls** menu.

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.

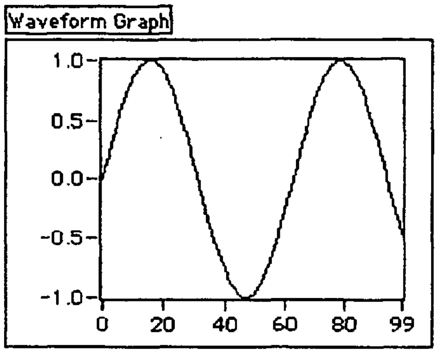
In addition to the information in this chapter, you might find study of the graph and chart examples included with LabVIEW to be helpful.

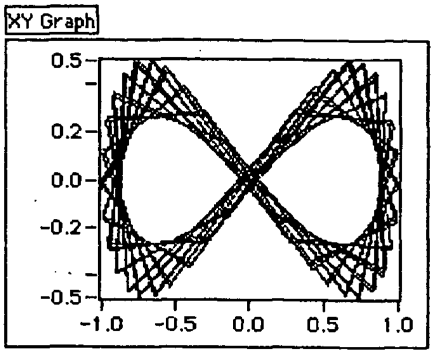
LabVIEW has two kinds of graphs and three kinds of charts.



Waveform and XY Graphs

You can obtain a *waveform graph* and *XY graph* from the **Graph** palette of the **Controls** menu. 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 amount of manipulation you need to do to your data before displaying it. These data types are described in the next section.

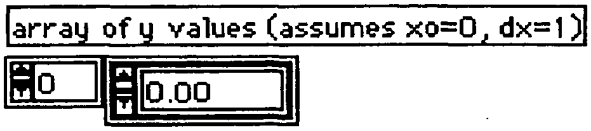
Following that are sections describing some of the more advanced options of the graph, including customizing the graph's appearance, displaying and manipulating cursors, and displaying a legend for the graph.

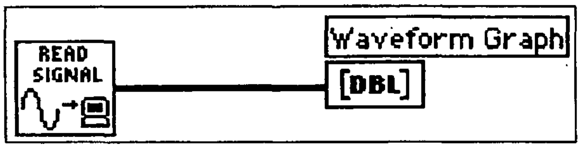
Creating a Single-Plot Graph

Waveform Graph Data Types

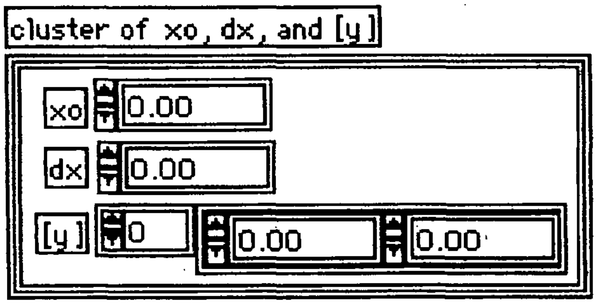
For single-plot graphs, the waveform graph accepts two data types. These data types are described in the following paragraphs.

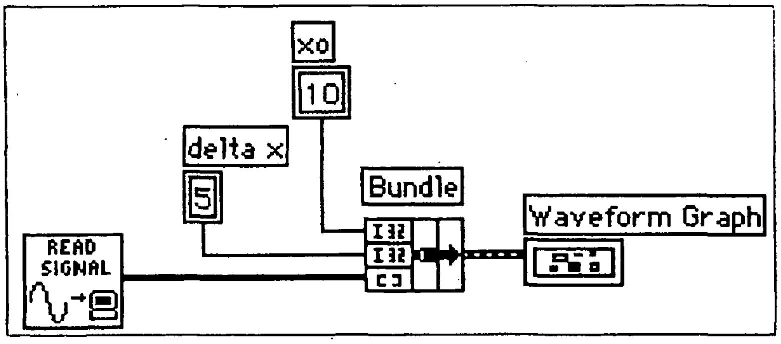
The first data type that the waveform graph accepts is a single array of values. LabVIEW interprets the data as points on the graph and increments the points (starting at x=0) by 1. The following diagram illustrates how you might create this kind of data.





The second data type is a cluster of an initial x value, a Δx value, and an array of y data. The following diagram illustrates how you might create this kind of data.

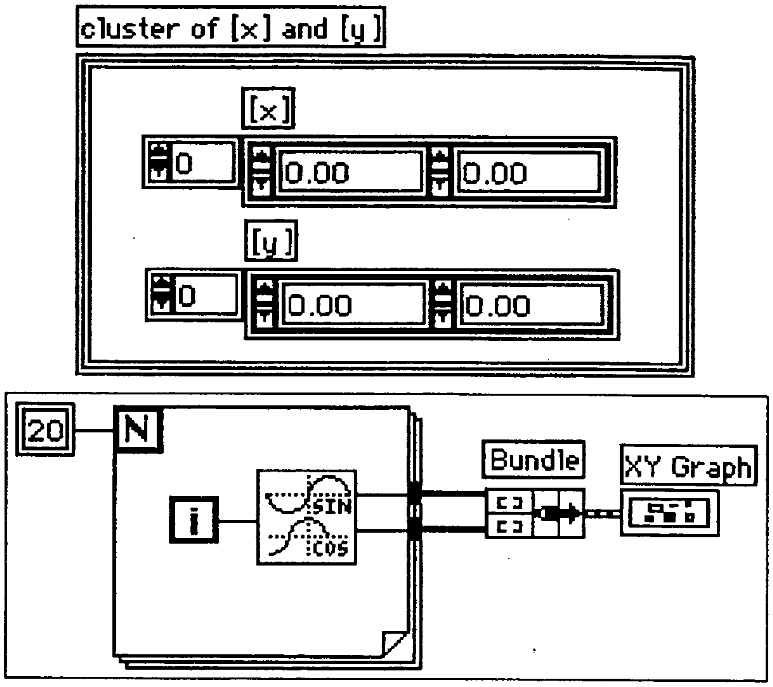




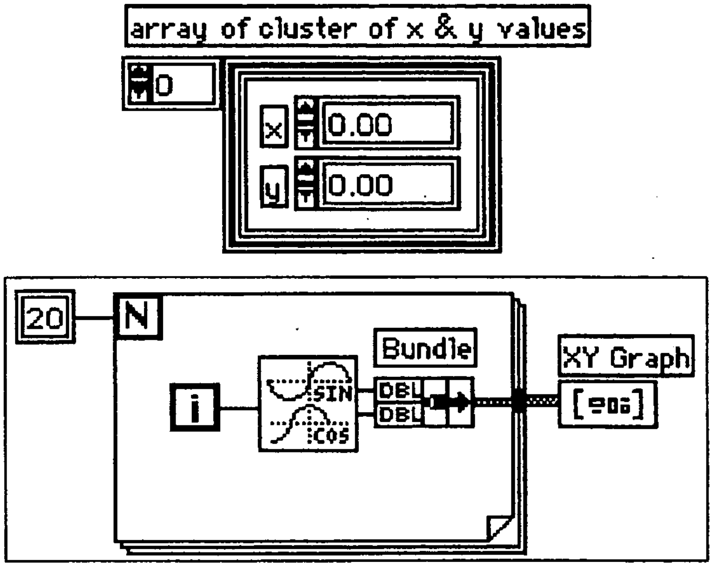
XY Graph Data Types

The XY graph accepts two data types for a single-plot. 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.



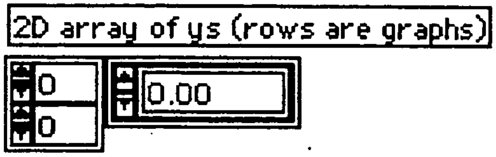
Creating a Multiplot Graph

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 the previous section 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.

Waveform Graph Data Types

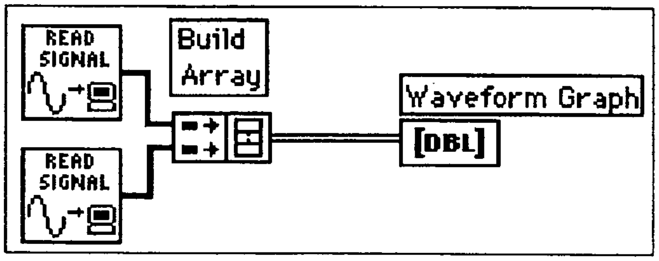
The multiplot waveform graph accepts five data types, as described in the following paragraphs.

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.

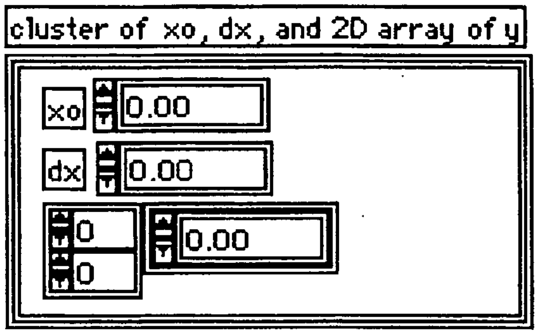


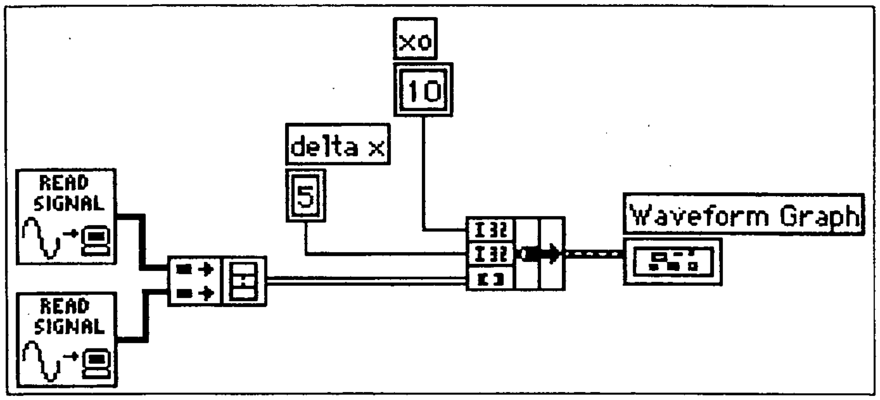
If you select the **Transpose Array** option from the graph pop-up menu, each column of data is treated as a plot. This is particularly useful when sampling multiple channels from a data acquisition board, because that data is returned as 2D arrays, with each channel stored as a separate column.

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.

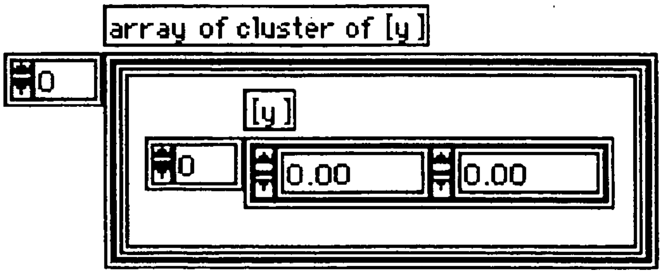


The second data type is a cluster of an x value, a Δx value, and a two-dimensional array of y data. The y data is interpreted as described for the previous data type. This data type is useful for displaying multiple signals that were all sampled at the same regular rate. The following diagram illustrates how you might create this kind of data.



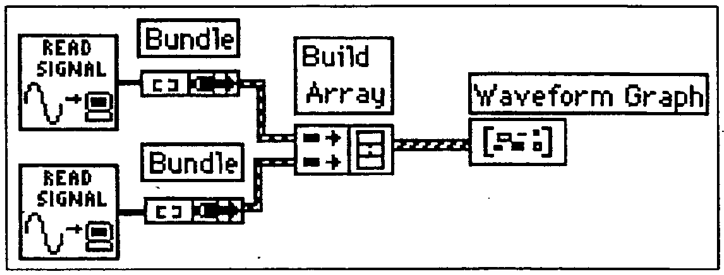


The third data type is an array of clusters of an array of y data. LabVIEW interprets this data as points on the graph, with the points starting at x=0 and incremented by 1.



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.

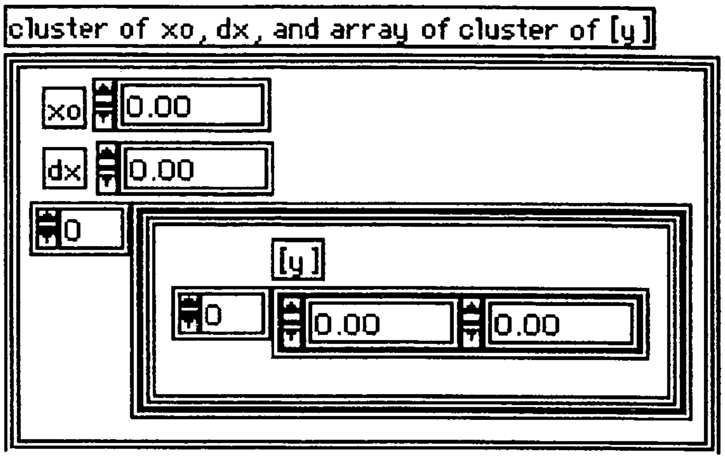
The following illustration shows how to create the appropriate data structure from two arrays.

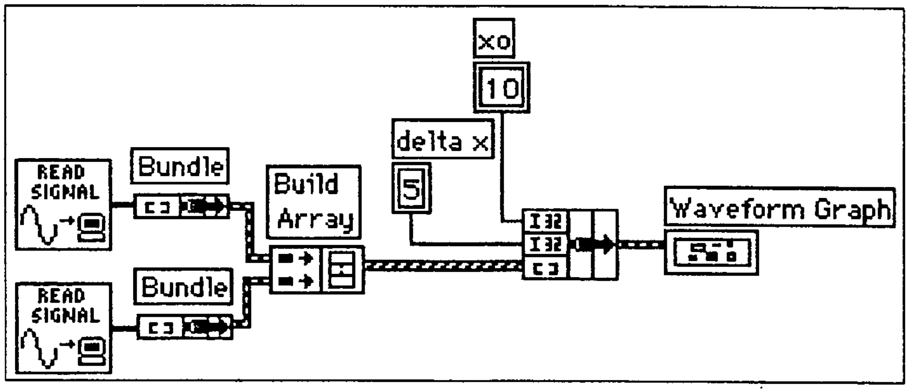


Another way to make arrays elements of an array of clusters is to use the build cluster array function

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.

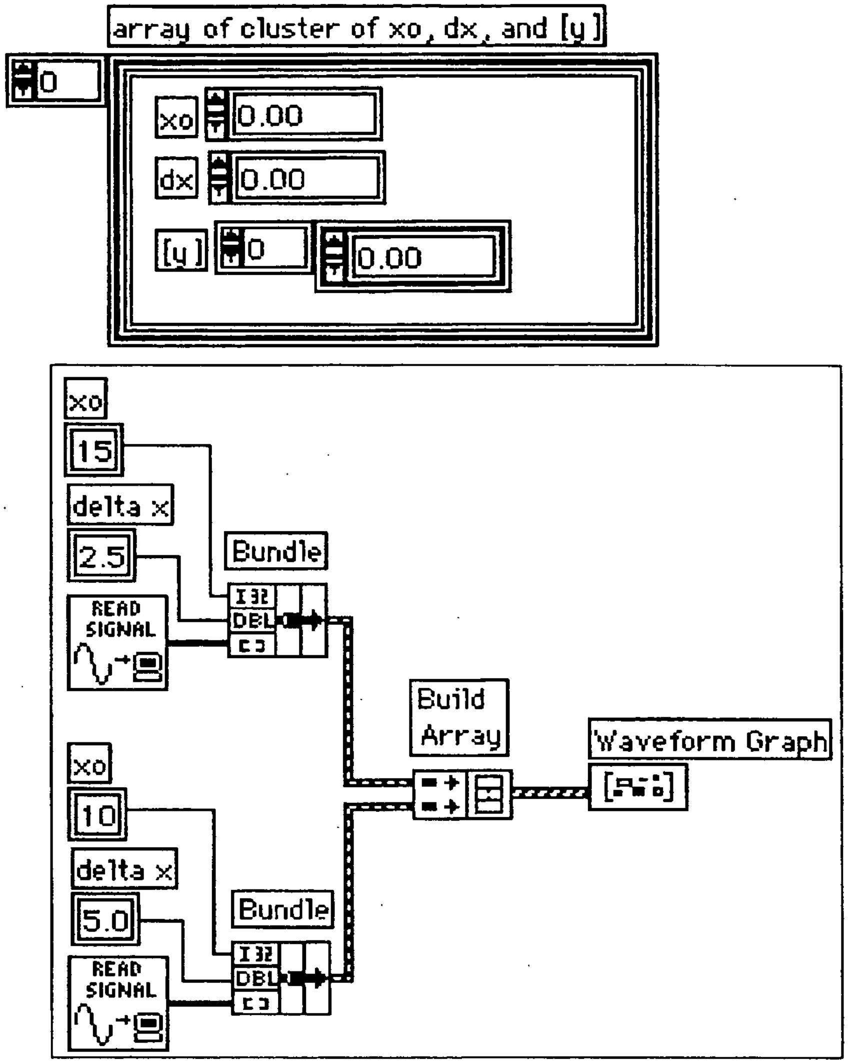
Use this data structure instead of a two-dimensional array if the number of elements in each plot is different. For example, you might need to sample data from several channels, but not for the same amount of time from each channel. You would then use this data structure because each row of a two-dimensional 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.

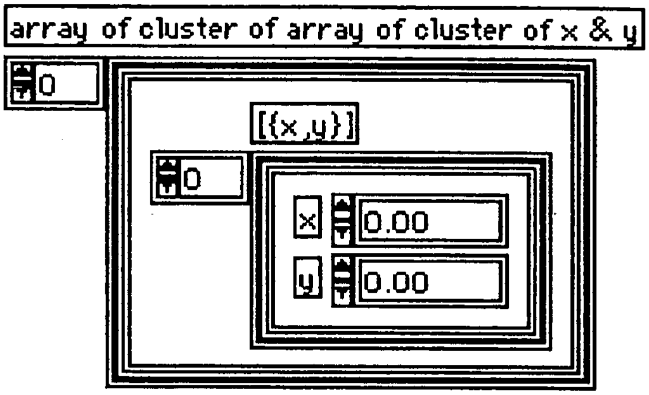
The fifth data type is an array of clusters of an x value, a Δx value, and an array of y data. This is the most general of the waveform graph multiplot data types, because it allows you to specify a unique starting point and increment for the X axis of each plot. The following diagram illustrates how you might create this kind of data.

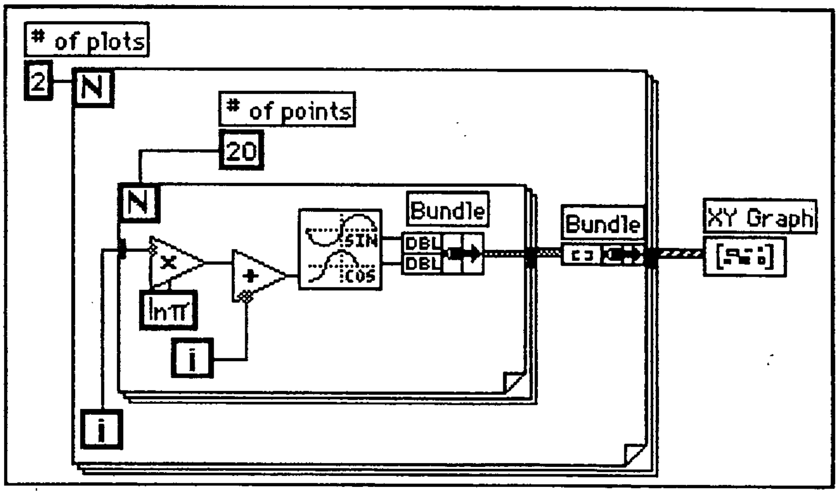


XY Graph Data Types

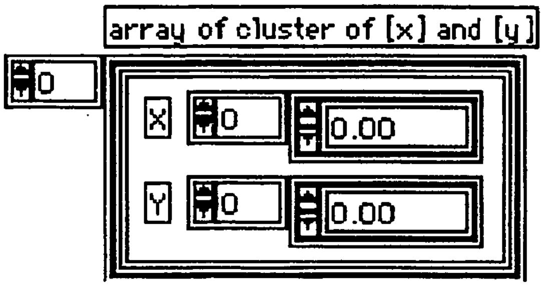
The XY graph accepts two multiplot data types, as described in the following paragraphs. Both of these data types are arrays of clusters of the single plot data types given earlier.

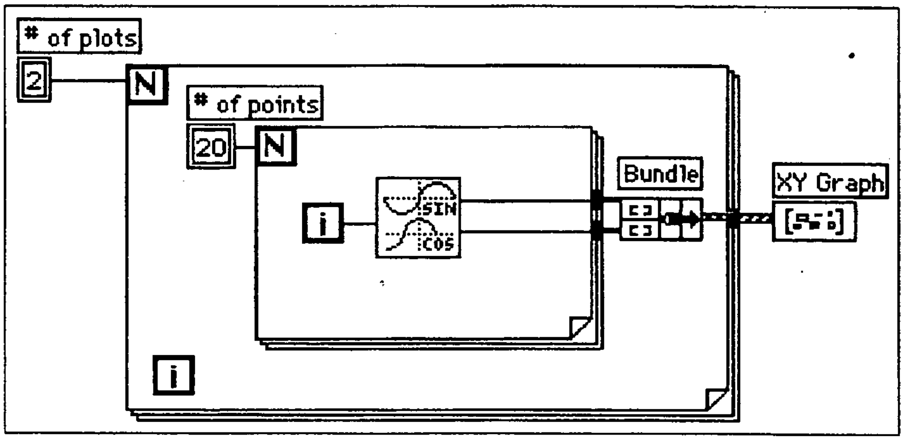
The first data type is an array of clusters of plots, where a plot is an array of points. A point is defined as a cluster containing an x and y value. The following diagram illustrates how you might create this kind of data.





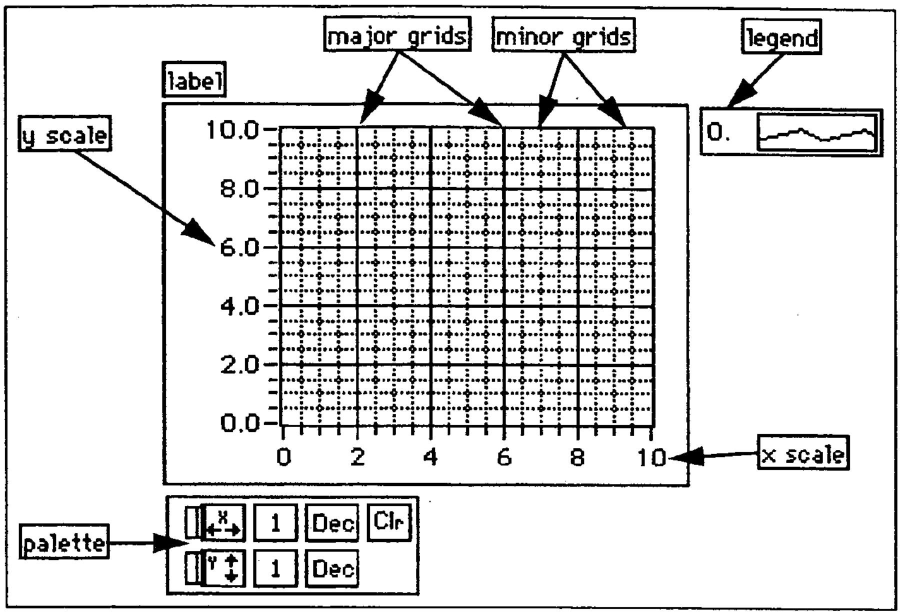
The second data type is an array of plots, where a plot is a cluster of an x array and a y array. The following diagram illustrates how you might create this kind of data.



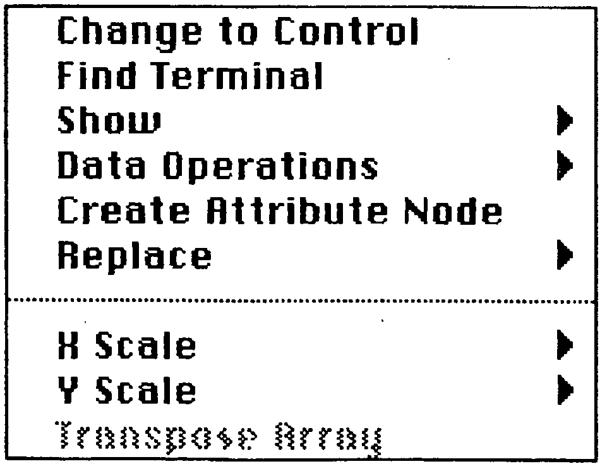


Graph Options

Both graphs have optional parts that can be shown or hidden using the **Show** submenu of the graph's 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 the *Graph Cursors* section.



Graphs have many options you can use to customize your data display. The graph pop-up menu is shown below. **Transpose Array** is available with the waveform graph only.

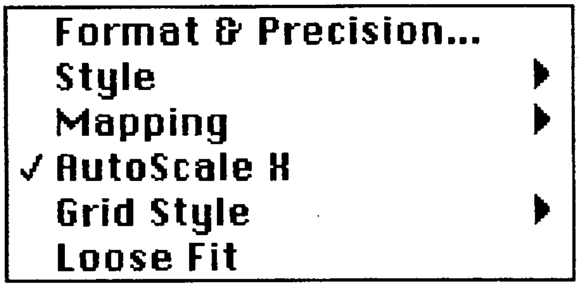


Scale Options

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 graph's pop-up menu. You can also control these autoscaling features from the graph's palette, as described later in this chapter. Autoscaling on is the default setting for LabVIEW. However, Autoscaling may cause the graph to be slower, depending upon the computer and video system you use.

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 X and Y scales each have a submenu of options, as shown in the following illustration.



The **Format & Precision...** options let you change the format and precision of the scale markers for each axis.

The **Style** palette lets you 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. This palette also lets you select whether you want the markers for a given axis to be visible.

The **Mapping** option of the scale menus lets you select whether the scale should display data using a linear or a logarithmic scale.

Use **AutoScale** to turn the autoscaling option on or off.

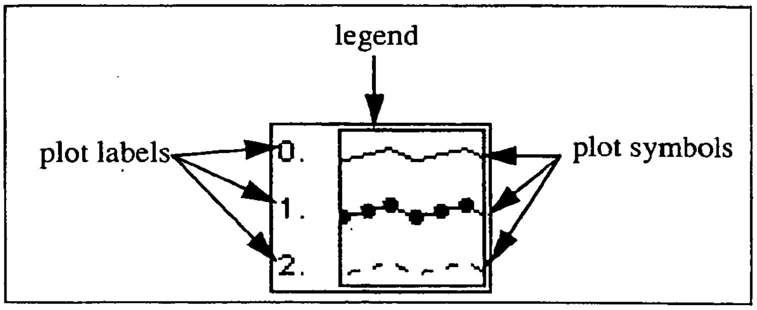
The **Grid Style** option lets you select whether you want no gridlines, gridlines only at major tick mark locations (points corresponding to scale labels) or major and minor tick marks (minor tick marks are interior points between labels).

Normally, the scales are set to exactly the range of the data when you perform a fit operation. You can use the **Loose Fit** option if you want LabVIEW to round the scale to "nicer" numbers. With a loose fit, the numbers are rounded to a multiple of the increment used for the scale. For example, if the markers increment by 5, then the minimum and maximum values are set to a multiple of 5.

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 be slower, depending on the computer and video system you use.

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.

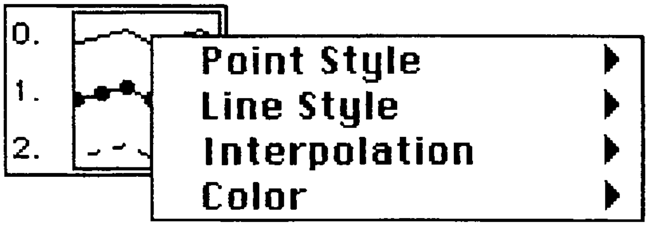


When you select **Legend**, only one plot appears. You can create more plots by dragging down a corner of the legend with the Resizing tool. After you set plot characteristics, LabVIEW assigns those characteristics to the plot, regardless of the whether the legend is visible. If the graph receives more 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 0. 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 will be displayed with the characteristics you assign it in the graph legend.

The plot sample pop-up menu is shown below.



The **Point Style** and **Line Style** options display styles you can use to distinguish a plot.

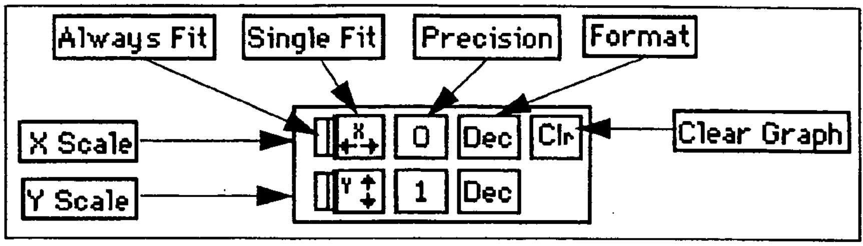
The **Interpolation** option determines how LabVIEW draws lines between plotted points. The **Line** option draws a straight line between plotted points. The **None** option does not draw a line, making it suitable for a scatter plot. The **Stepped** option links points with a right-angled elbow. You can use this option to create histogram-like plots.

The **Color** option displays the palette for selecting the plot color. You can also color the plots on the legend with the Coloring tool, and you can change the plot colors while in run mode.

Use the **Defaults** option to change the text attributes for the legend.

Using the Palette

With the palette, you can access several useful functions while the VI executes. You can reset the graph, scale the X or Y axis, and change the display format of the scale at any time. The palette, which you access from the **Show** menu of the graph pop-up menu, is shown in the following illustration.



Clr
clear graph button

The clear graph button clears all data from the graph. This function is also available from the run-time pop-up menu.

The rest of the palette is organized in two rows, one for the X axis and the other for the Y axis.

X
Y
fit switches

The fit switches are locking switches that control whether the graph automatically changes the display area or scales to display current or new data. Clicking on the body of the switch performs a single fit operation. Clicking on the lock (the smaller rectangle on the left of the switch) controls whether LabVIEW locks the scale on continually fits new data. In autoscale mode, the graph is locked by default. You can turn this off by clicking on the lock part of the switch. You can also turn on the autoscale mode from the run-time pop-up menu and from the **X scale** and **Y scale** submenus.

0
set precision

To the right of the fit switch is a field that allows you to adjust the number of digits the scale displays. Use the Operating tool to change this setting.

Dec
set format

You can switch the scale readouts to represent decimal, scientific, or engineering format by clicking on the button with the Operating tool.

Waveform Chart

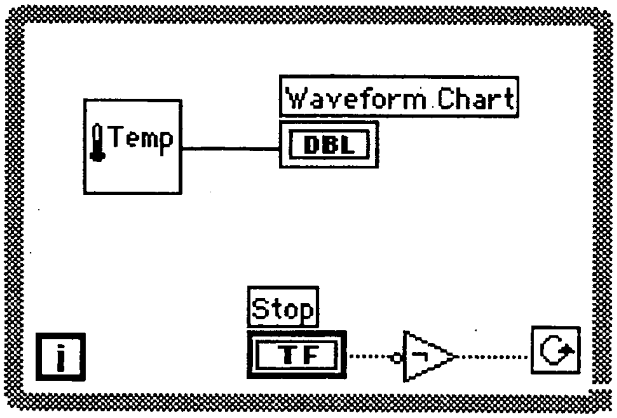
The waveform chart is a special type of numeric indicator that displays one or more plots. Charts are different from graphs in that charts retain previous data, up to a maximum which you can define. New data is appended to the old data, letting you see the current value in context with previous data.

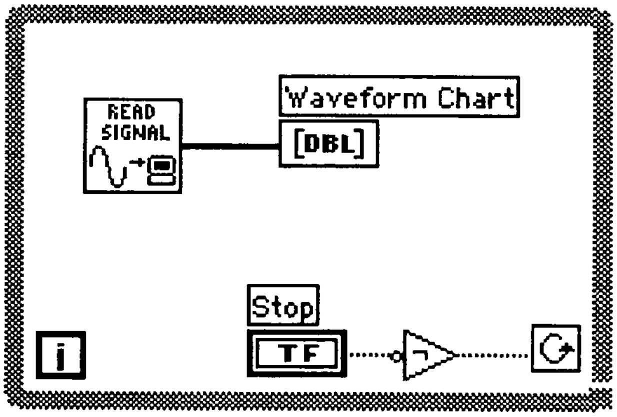
Waveform Chart Data Types

You can pass charts either a single value or multiple values at a time. As with the waveform graph, each value will be 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.

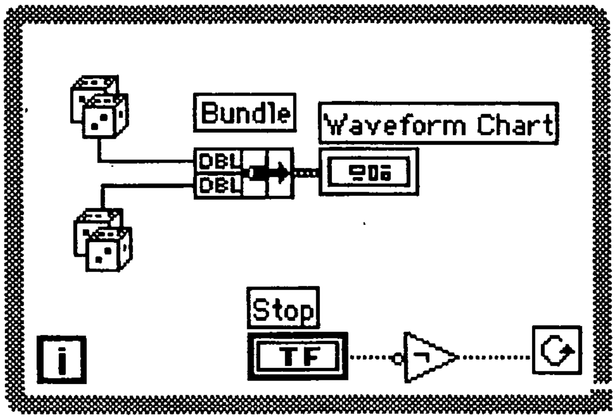
Following are diagrams illustrating how you can use the chart for each of these kinds of data.



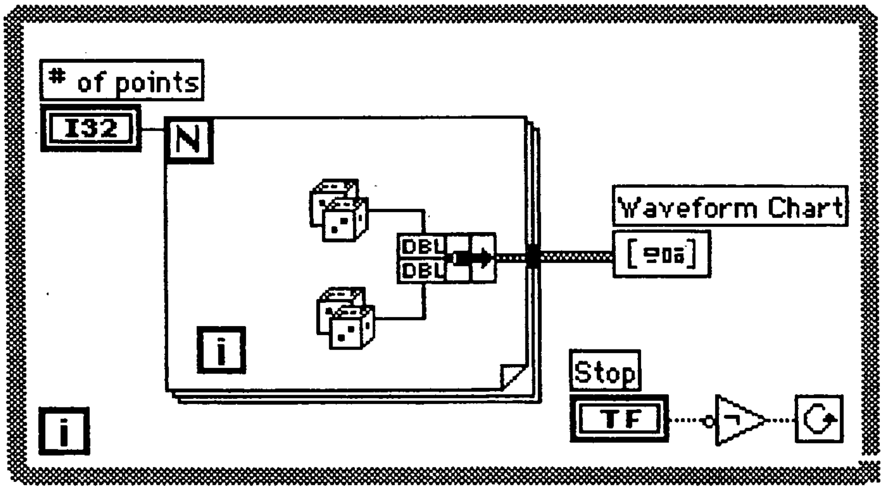


You get the best performance from a chart when you pass it multiple points at a time, because the chart has to be redrawn only once for each waveform instead of once for each point.

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 below.

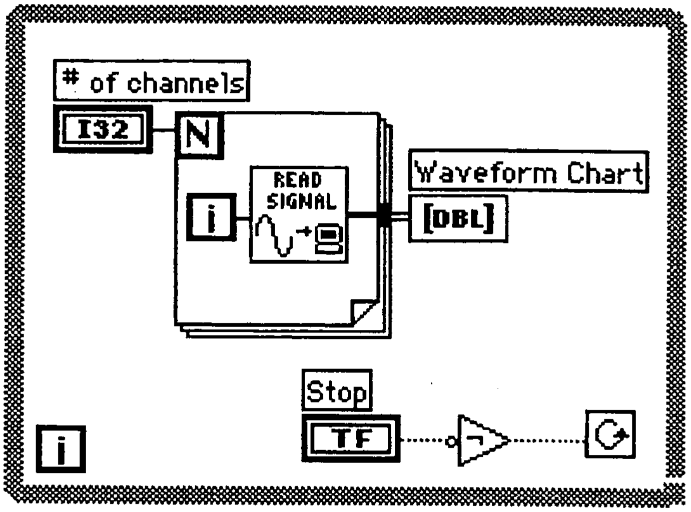


If you want to pass multiple points for plots in a single update, you can wire an array of clusters of numerics to the chart. Each numeric represents a single point for each of the plots. An example of this is shown below.



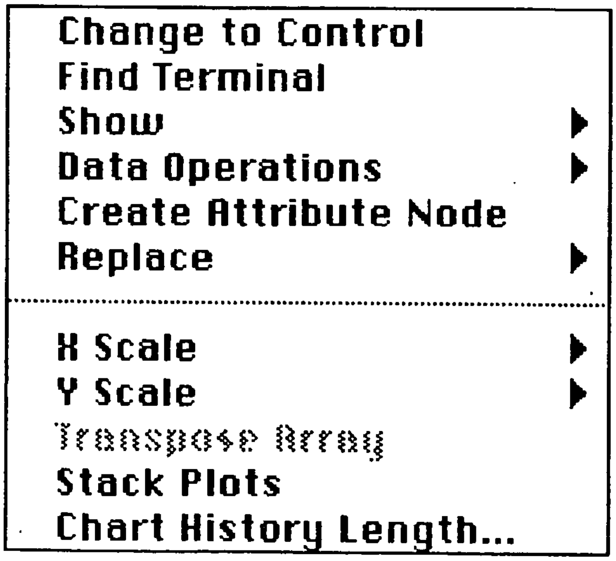
If the number of plots you want to display cannot be determined until runtime, or you want to pass multiple points for plots in a single update, then you can wire a two-dimensional array of data to the chart. As with the waveform graph, rows are normally treated as new data for each plot.

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 the *Scale Options*, *Using the Legend*, and *Using the Palette* sections of this chapter for more information.) The waveform chart does not have support for cursors. The following illustration shows the chart pop-up menu.



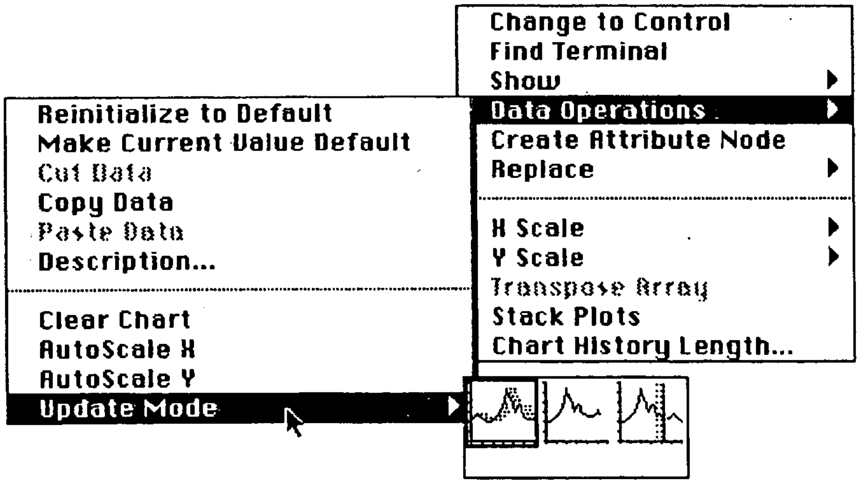
The **Show** submenu of the chart's pop-up menu also lets you show or hide optional digital display(s) and a scroll bar. The digital display option displays the latest value being plotted. The last value passed to the chart from the diagram is the latest new value for the chart. There is one digital 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 scroll bar, or by changing the x scale range to view old data.

The chart has a limit to the amount of data it will 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 1024 points. You can change the length of this buffer using the **Chart History Length...** option from the chart's pop-up menu.

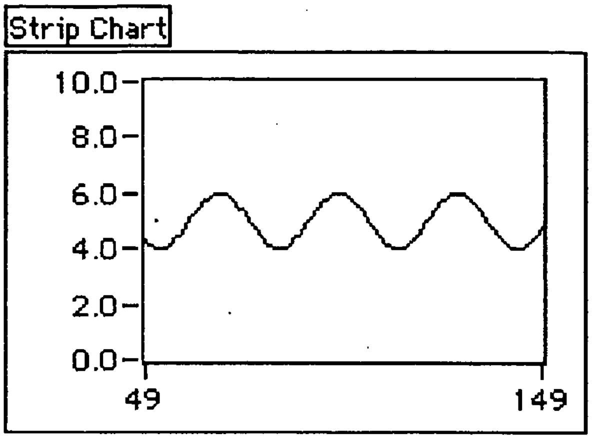
Chart Update Modes

The **Update Mode** option from the **Data Operations** submenu of the chart's pop-up menu lets you change the way the chart behaves when new data is added to the display.

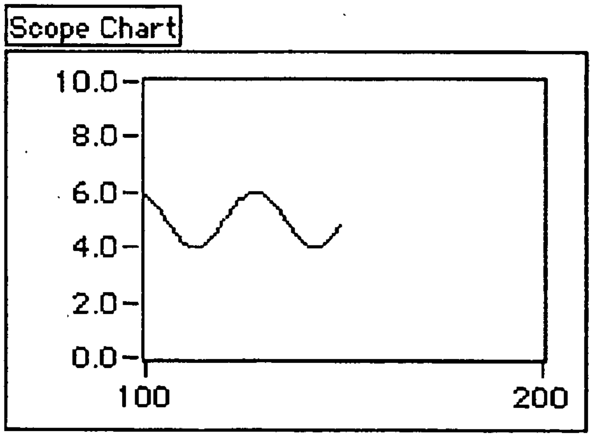


The three options, strip chart, scope chart, and sweep chart are illustrated in the following pictures, and described in the subsequent paragraphs. The default mode is strip chart.

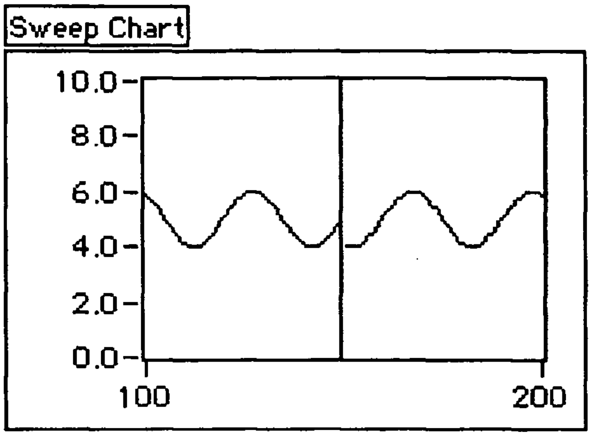
The *strip chart* mode has a scrolling display similar to a paper tape strip chart recorder. As each new value is received, it is plotted at the right margin, and old values shift to the left.



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, and 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.

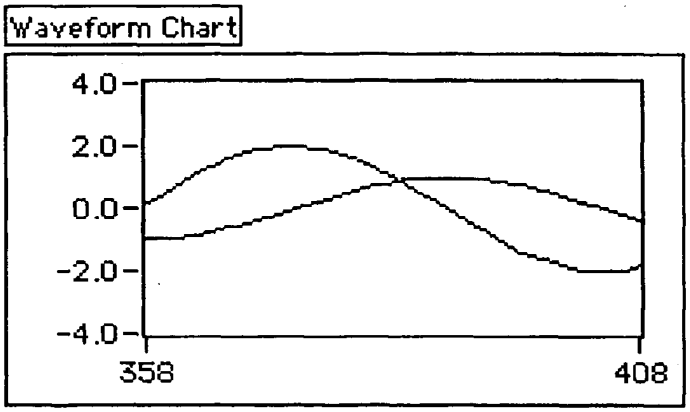


The *sweep chart* mode acts much like the scope chart, but it does not blank when the data hits the right border. Instead, a moving vertical line marks the beginning of new data and moves across the display as new data is added.

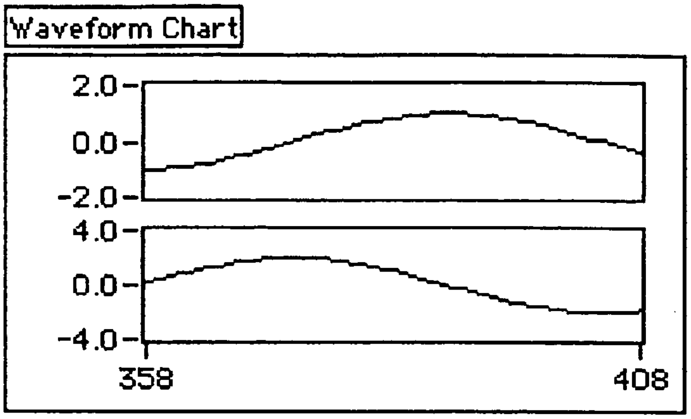


Stacked versus Overlaid Plots

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.



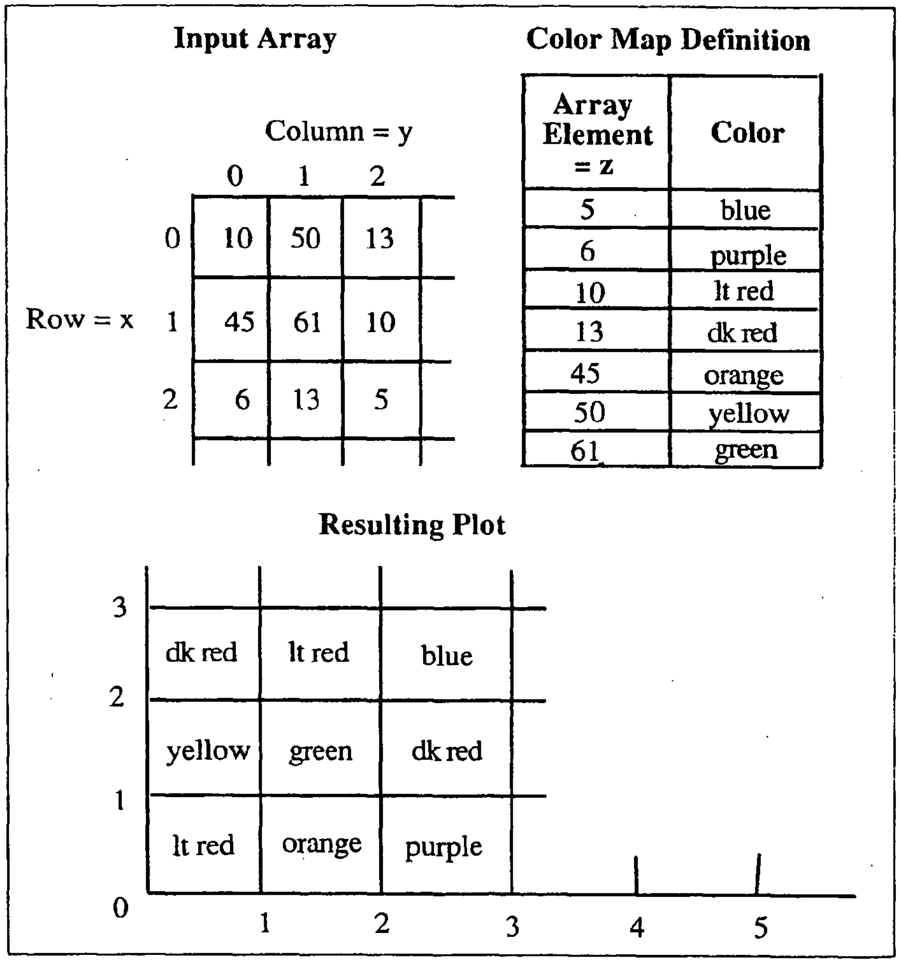
You can alternatively display multiple plots by stacking onc above the other, with a different Y scale for each plot, by selecting the **Stack Plots** option from the chart pop-up menu. If you do this, then each chart's Y scale can have a different range. An example of stacked plots is shown in the following 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.



Input Array

Row = x	Column = y 0	1	2
0	10	50	13
1	45	61	10
2	6	13	5

Color Map Definition

Array Element = z	Color
5	blue
6	purple
10	lt red
13	dk red
45	orange
50	yellow
61	green

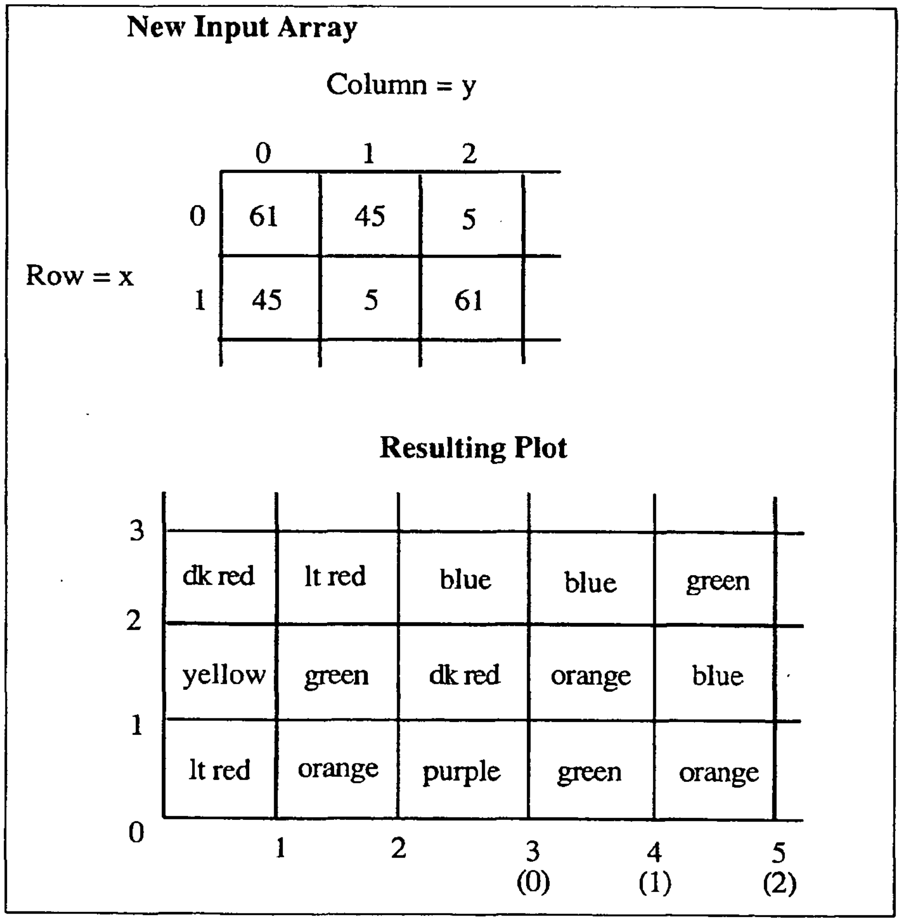
Resulting Plot

2–3	dk red	lt red	blue
1–2	yellow	green	dk red
0–1	lt red	orange	purple

You can define the colors for the intensity chart interactively, by using an optional color ramp control, or programmatically through the chart attribute node. The *Defining the Color Mapping* section later in this chapter 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 discreet colors.

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.



	0	1	2
0	61	45	5
1	45	5	61

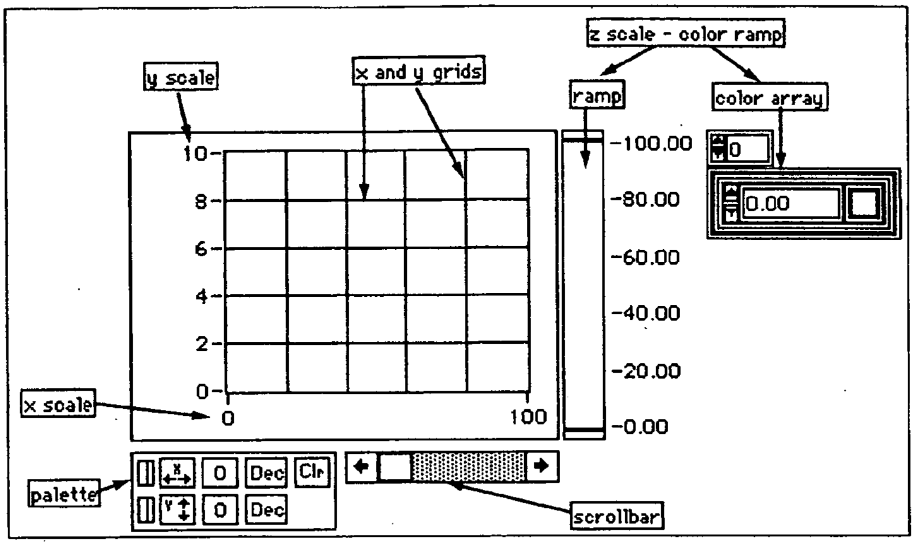
	1	2	3 (0)	4 (1)	5 (2)
3–2	dk red	lt red	blue	blue	green
2–1	yellow	green	dk red	orange	blue
1–0	lt red	orange	purple	green	orange

When the chart display is full, the oldest data scrolls of the left side of the chart.

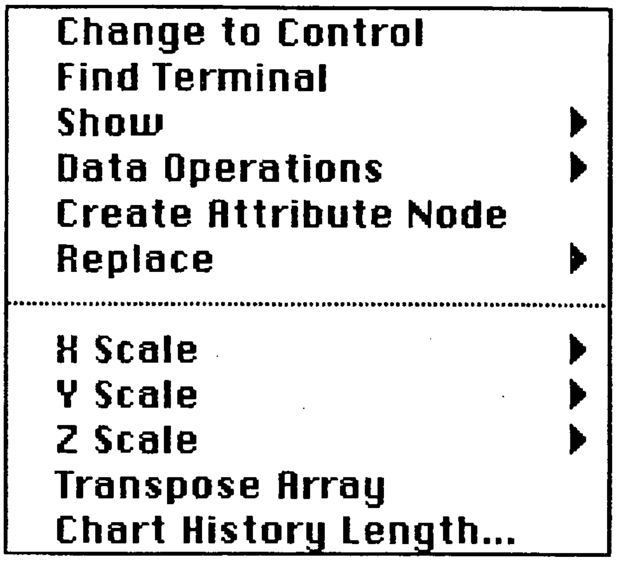
Intensity Chart Options

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's 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 which is 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 the *Graph Cursor* section.



Intensity charts have many options you can use to customize your data display. The intensity chart pop-up menu is shown below.



Most of these options are identical to the options for the waveform chart. The **Show** menu lets you show and hide the z scale's color ramp and color array. The **X Scale** and **Y Scale** menus are identical to the corresponding menus for the waveform chart. The **Z Scale** option is almost the same, except it doesn't have the grid and loose fit options.

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, you should increase the amount of memory configured for LabVIEW's use.

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 control in Chapter 5, *Numeric Controls and Indicators*, 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 ramp. 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 scale of the color ramp. If the color ramp 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).

Intensity Graph

The intensity graph is essentially the same as the intensity chart, except it doesn't retain previous data. Each time it is passed new data, the new data replaces old data as it arrives.

Intensity Graph Data Type

The intensity chart accepts two-dimensional arrays of numbers, where each number will be 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's 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 scroll bar, and it does not have history options.

The intensity graph can have cursors like other graphs. Each cursor will display the X, Y, and Z values for a specified point on the graph. See the *Graph Cursors* section of this chapter for more information on manipulating the graph's cursors.

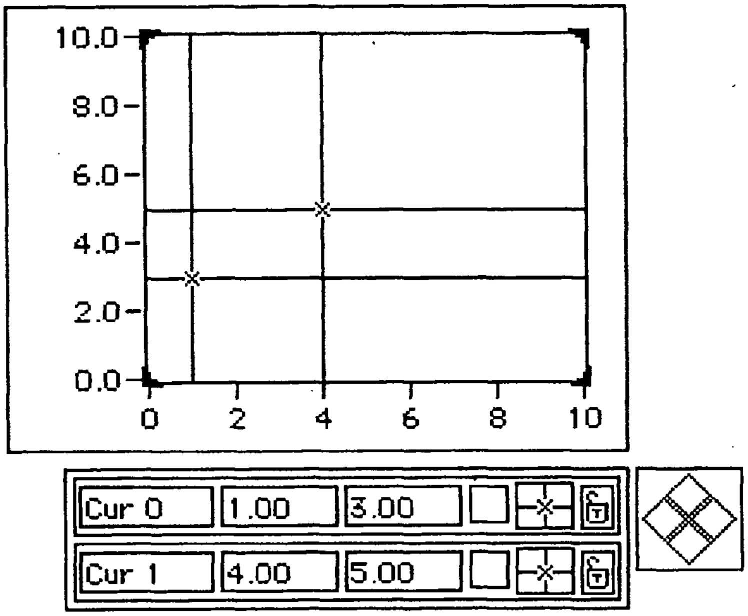
You set the color mapping in the same way that you set it for the intensity graph.

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. A cursor can also be used 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.

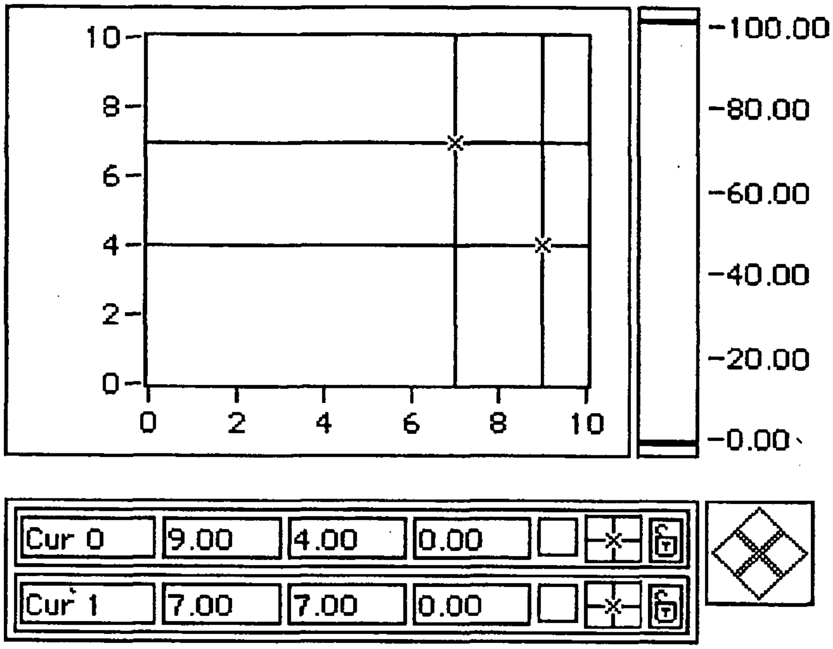
Following are illustrations of a waveform graph and an intensity graph with the cursor palette displayed.



Cur 0 1.00 3.00

Cur 1 4.00 5.00

Waveform Graph



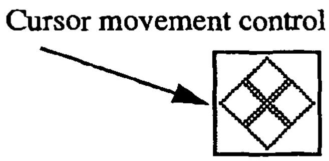
Cur 0	9.00	4.00	0.00	□	-✕-	🔓
Cur 1	7.00	7.00	0.00	□	-✕-	🔓

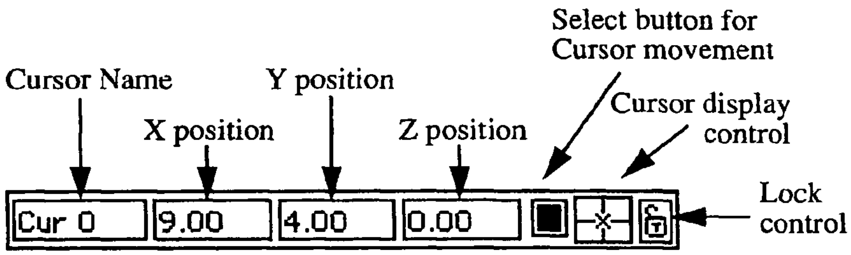
Intensity Graph

Each cursor for a graph has

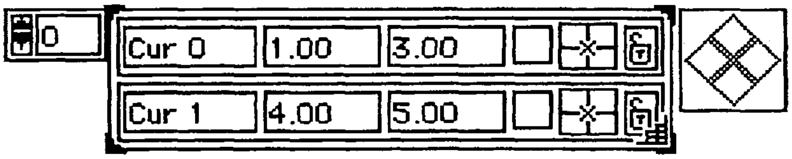
- A label
- X and Y coordinates, and Z coordinate, if applicable
- A button that marks the plot for movement with the plot cursor pad

- 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



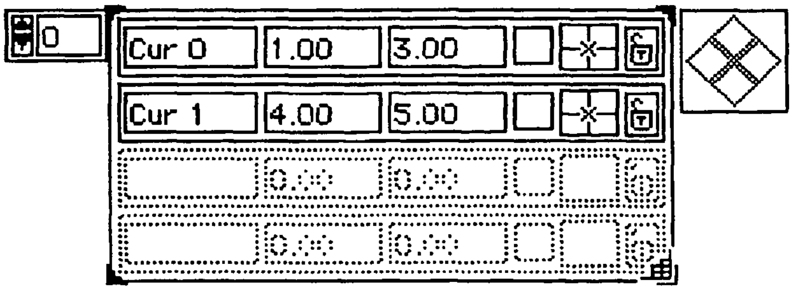


The cursor palette behaves like an array. You can stretch it to display multiple cursors, and you can use the index control to view other cursors in the palctte. Use the **Show** options on the pop-up menu to display the index control.



0

Cur 0	1.00	3.00
Cur 1	4.00	5.00

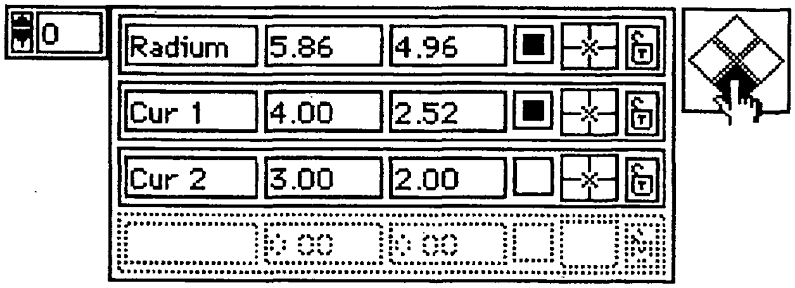


0

Cur 0	1.00	3.00
Cur 1	4.00	5.00
	0.00	0.00
	0.00	0.00

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 the *Selecting Array Cells* section of Chapter 8, *Array and Cluster Controls and Indicators*, for more information.

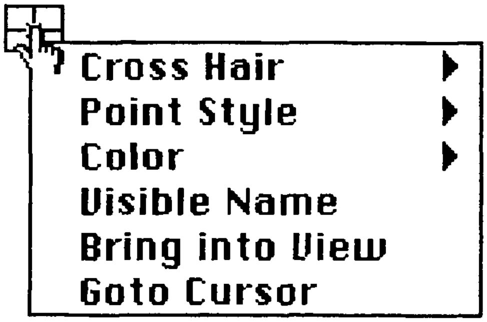
You can move a cursor on a graph or chart by dragging it with the operating tool, or by using the cursor movement control. Clicking the arrows on the cursor movement control causes all cursors selected to move in the specified direction. You select cursors either by moving them on the graph with the 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.



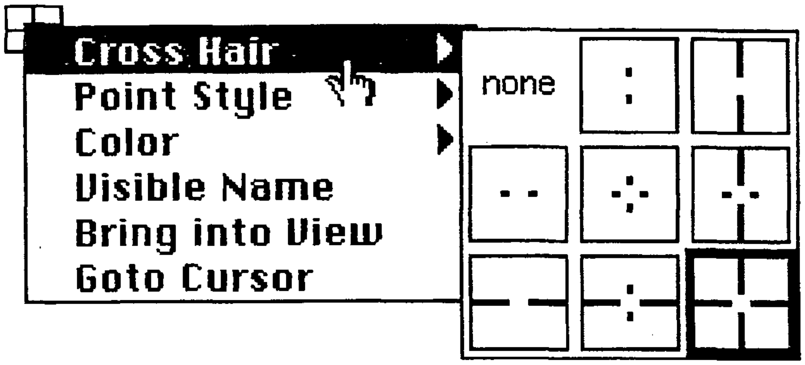
Radium	5.86	4.96
Cur 1	4.00	2.52
Cur 2	3.00	2.00

cursor display control

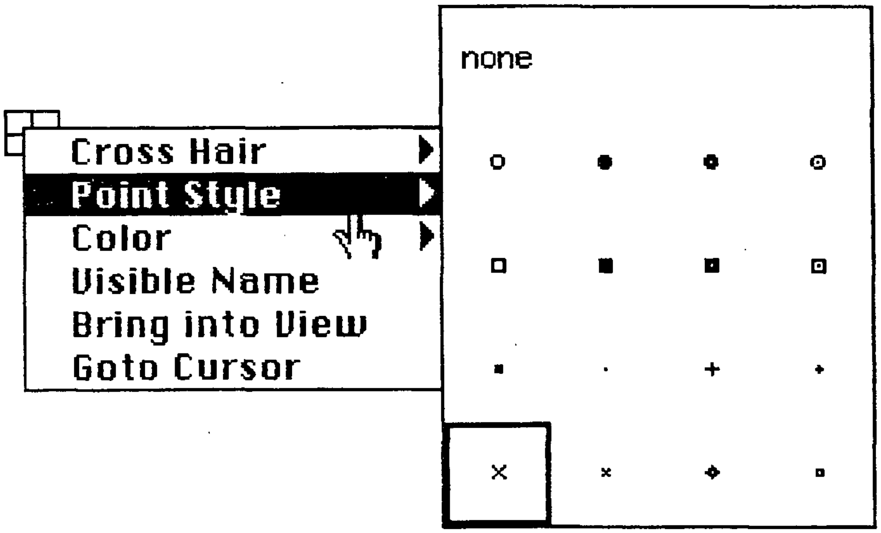
Clicking on the cursor display control with the operating tool displays a pop-up menu that you can use to control the look of the cursor and the visibility of the cursor name on the plot.



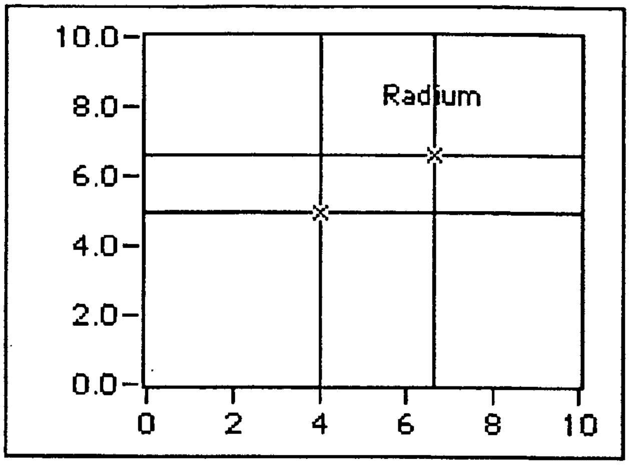
From this menu you can select the style of the cross hairs. The cross hairs can consist of a vertical and/or horizontal line extending to infinity, a smaller vertical and/or horizontal line centered about the cursor location, or no cross hairs, as shown in the following illustration.



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.



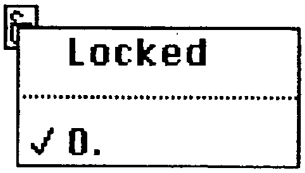
Selecting **Bring into View** moves the cursor back into the displayed region of the graph. This is helpful when the cursor has moved out of visible range. Selecting this option changes the `(x,y)` coordinate position of the cursor.

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.

unlocked locked

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 the a specific plot. If you lock the cursor onto a plot, the button changes to a closed lock.

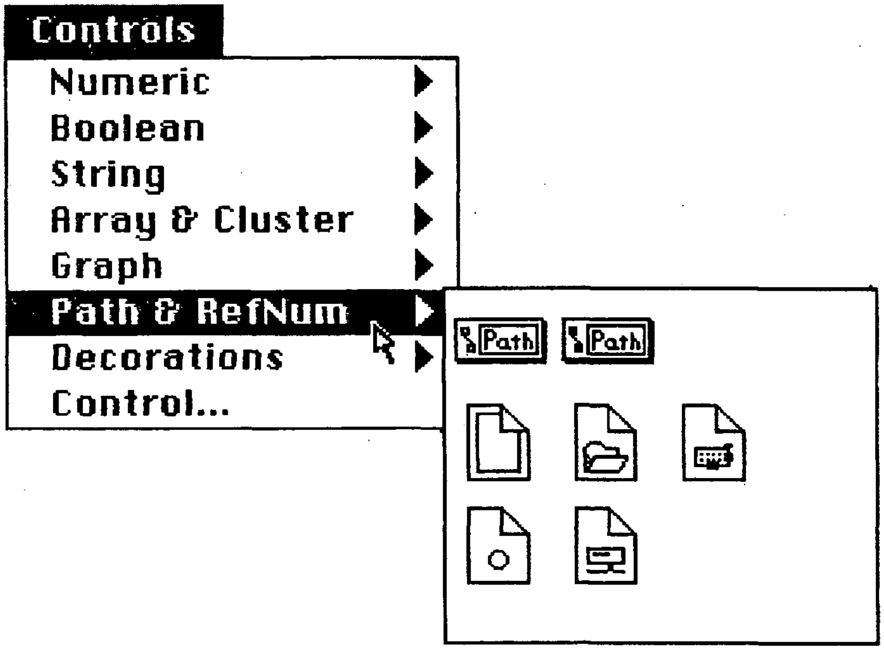


A large number of options are available for creating, controlling, and reading cursors or markers programmatically using the Attribute Node for a graph. See Chapter 15, *Attribute Nodes*, for further information.

Chapter 10

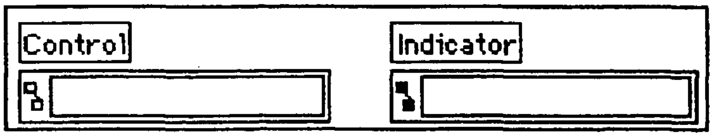
Path Controls and Refnums

This chapter describes how to use file path controls and refnums. Path Controls and Refnums are available from the **Path & Refnum** palette of the **Controls** menu.



Using Path Controls and Indicators

A path control and path indicator are shown in the following illustration.



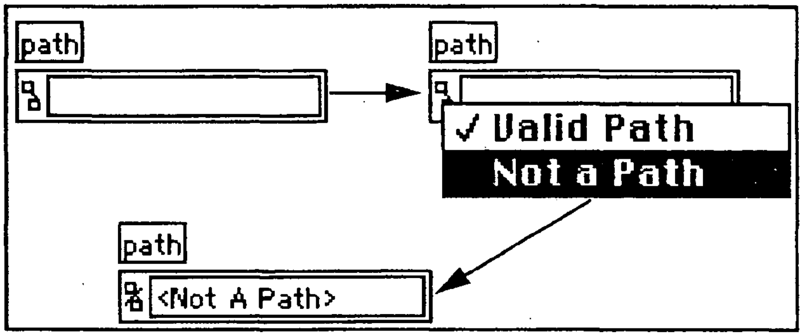
Path Symbol

Not A Path Symbol

The path control is used to enter a path as a file or directory. If a function that is supposed to return a path fails, it will return 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.

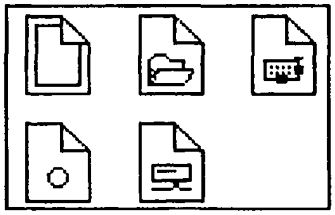
`NotAPath` is the canonical non-valid path, used to represent an invalid path. LabVIEW typically uses `NotAPath` as a subVI output or as a return value from a CIN.

When you are in run mode, 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. 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.



Refnums

The **Path & Refnums** palette from the **Controls** menu contains several **Refnum** controls, as shown in the following illustration. These refnums are generally used to identify I/O operations.

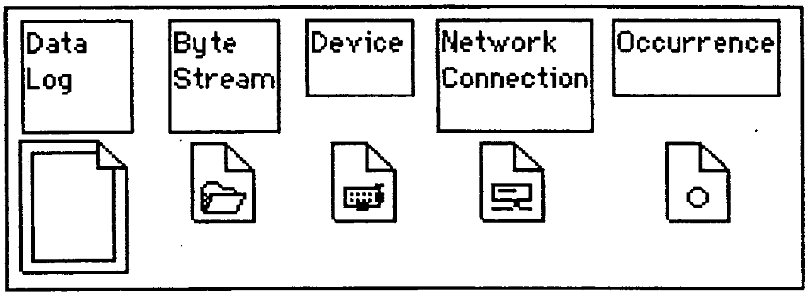


When you open a file, you specify the path of the file you want to open. Because LabVIEW allows you to open more than one file at a time, you need to specify which operations apply to which open file. The path control is not sufficient for this, because you can open one file more than once, concurrently.

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. When you close the file, the refnum is disassociated from it.

▶ Note: ***You rarely need to use a refnum control. You only need one if you want to pass a refnum between two VIs.***

LabVIEW has five refnum datatypes for various kinds of I/O. For each refnum, there is a corresponding control. The following illustration shows the different kinds of refnums.



There are two kinds of file refnums, one for data log files and one for byte stream files.

Because data log files have an inherent structure, the *data log refnum* is used to pass the refnum as well as a description of the file type to (or from) calling VIs. The data log refnum is resizeable, 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 data log 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. *Byte Stream refnums* are used with byte stream files, either text or non-record binary files. *Device refnums* are used with the device I/O functions. The *network connection refnum* is used with the TCP/IP VIs. The *occurrence refnum* is used with the occurrence functions.

Block Diagram Programming
Chapters 11 through 19

Chapters 11 through 19 Block Diagram Programming

Chapter 11

Introduction to the Block Diagram

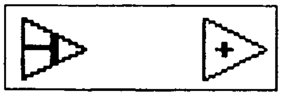
This chapter describes nodes, terminals, and wires–the elements you use to build a block diagram. The chapter also discusses how to get online help when you are putting these items together.

Block Diagram

You create block diagrams with nodes, terminals, and wires.

Nodes are program execution elements. They are analogous to statements, operators, functions, and subroutines in conventional programming languages.

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**.



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

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 Figure 11-1, *LabVIEW Control and Indicator Terminal Symbols.* 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 copy or delete a terminal. LabVIEW automatically creates or disposes of terminals when you create or remove front panel controls or indicators.

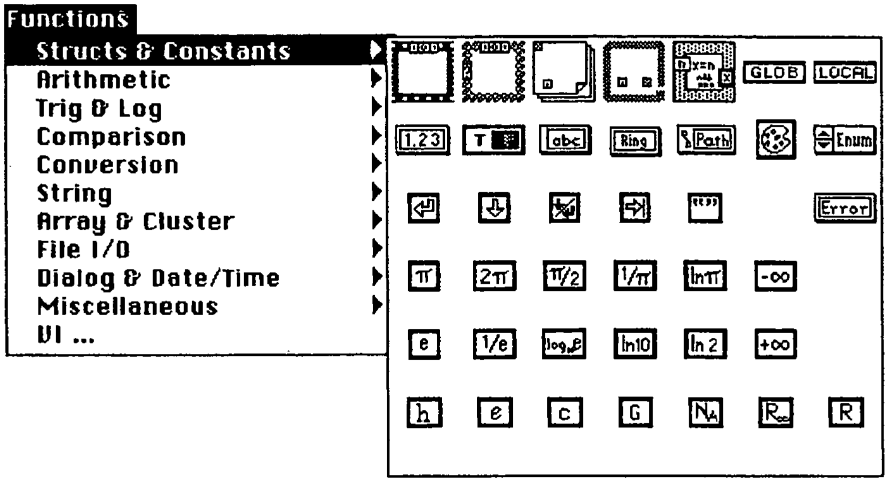
An array terminal encloses one of the data types shown in square brackets, and takes on the color of the data type of the array.

Figure 11-1. LabVIEW Control and Indicator Symbols.

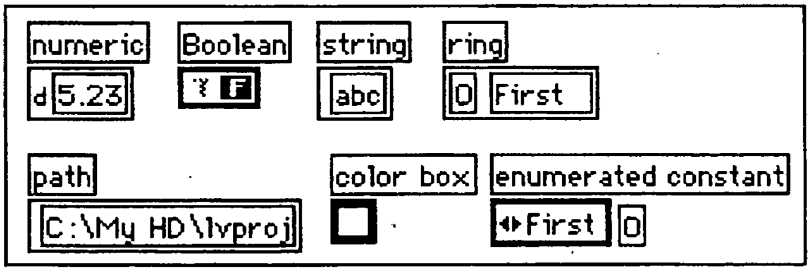
Control	Indicator	Description	Color
EXT	EXT	Extended-precision floating-point	Orange
DBL	DBL	Double-precision floating-point	Orange
SGL	SGL	Single-precision floating-point	Orange
CXT	CXT	Complex extended-precision floating-point	Orange
CDB	CDB	Complex double-precision floating-point	Orange
CSG	CSG	Complex single-precision floating-point	Orange
U32	U32	Unsigned 32-bit integer	Blue
U16	U16	Unsigned 16-bit integer	Blue
U8	U8	Unsigned 8-bit integer	Blue
I32	I32	32-bit integer (long word)	Blue
I16	I16	16-bit integer (word)	Blue
I8	I8	8-bit integer	Blue
▭	▭	Cluster	Brown
[SGL]	[SGL]	Array	Varied
◀▶	◀▶	Enumeration	Blue
⊸	⊸	Path	Aqua
🗋	🗋	Refnum	Aqua

User-Defined and Universal Constants

Constants are terminals on block diagram that supply data values directly to the block diagram. You set the value of a *user-defined constant* prior to program execution, and you cannot change its value during execution. *Universal constants* have fixed values. You choose these constants from the **Structs & Constants** palette of the **Functions** menu, shown below.



The user-defined constants located in the second row in palette, 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. The following illustration shows the types of constants.



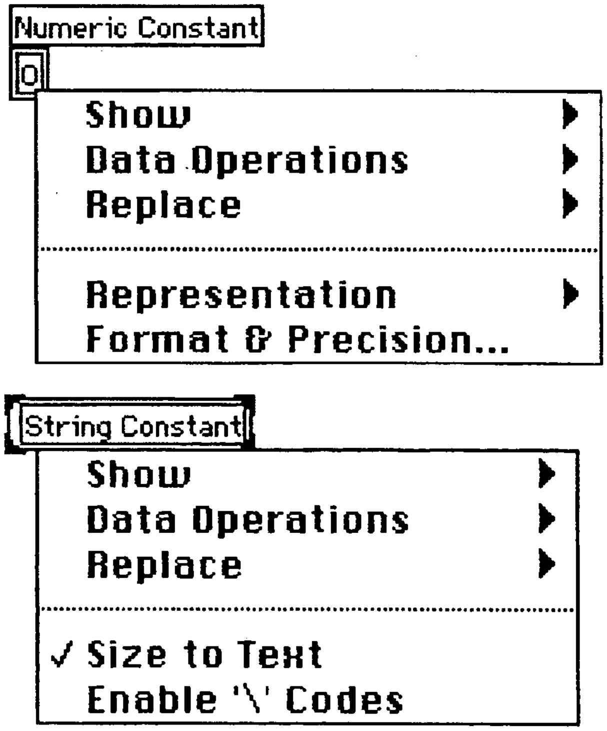
The ring constant associates text with a number, the same way a ring control on the front panel does. The value of the ring constant is an unsigned 16-bit integer. Although you can change the representation of a ring constant to any numeric type except complex, the value will still always be an integer from 0 to *n*-1.

The enumeration constant is similar to the ring constant except that the mnemonics 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.

You can use the path constant to create a constant path value on the diagram.

You can select colors from the color box constant to use with the color box control, an internal control whose values correlate to specific colors. Set the color box by clicking on it with the Coloring tool and choosing the color you want from the color palette.

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 below.



The path, string, ring, enumeration, and color box constants are resizable, while the boolean and numeric constants are not. When you first place most constants on the diagram, LabVIEW highlights the text box, indicating that you can type into it immediately without changing the tool. This does not take place with boolean, color box, or fixed constants. 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 character constants include five commonly used nondisplayable string characters and an empty string. Table 11-1 shows the symbols and names for these constants.

Table 11-1. Universal Character Constants

Icon	Name
↵	Carriage return
⇩	Line feed
⇥	Tab
“”	Empty String
⤦	End of Line

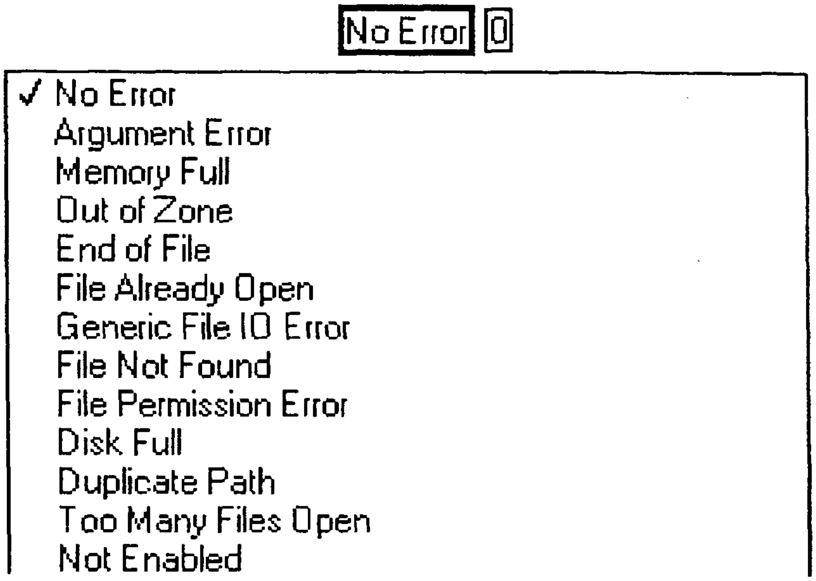
Universal numeric constants are a set of high-precision and commonly used mathematical and physical values, such as pi (π) and the speed of light (c). Table 11-2 lists these constants, whose values reflect the precision of the extended-precision floating-point number in LabVIEW or the precision to which they are known, in the case of physical constants.

Table 11-2. Universal Numeric Constants

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	- ∞
+∞	Positive Infinity	∞
e	Natural Logarithm Base	2.71828182845904520
1/e	Reciprocal of e	0.36787944117144232
$\log_{10}e$	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 m^2/kg^2)	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

The Help window, discussed later in this chapter, shows the value of a universal constant when you touch the constant with any tool.

The error 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 figure. This ring is useful with the file I/O functions, because it makes the diagrams more descriptive. For instance, 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).



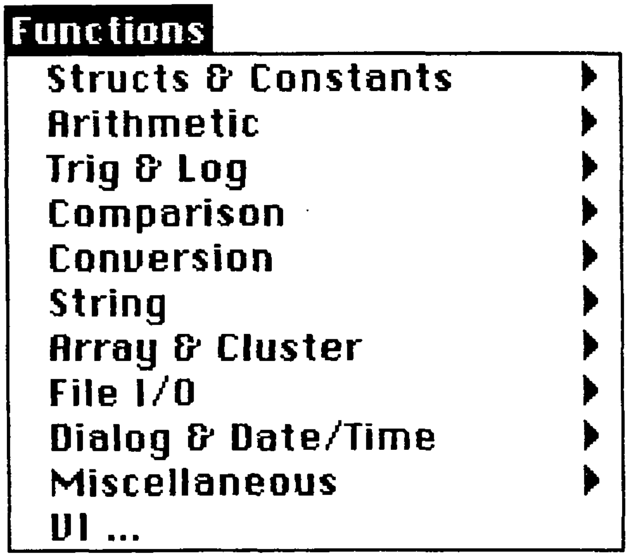
Nodes

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. They are explained in the following sections. CINs are interfaces between the block diagram and code you write in conventional programming languages such as C or Pascal. See the *LabVIEW Code Interface Reference Manual* for more information. Structures supplement the LabVIEW dataflow programming model to control execution order. They are discussed briefly in a section that follows, but see Chapter 13, *Structures*, for in-depth information. Formula Nodes supplement the functions by allowing you to use formulas on the block diagram. See Chapter 14, *The Formula Node*, for more information. Attribute Nodes let you change control attributes programmatically. See Chapter 15, *Attribute Nodes*, for more information.

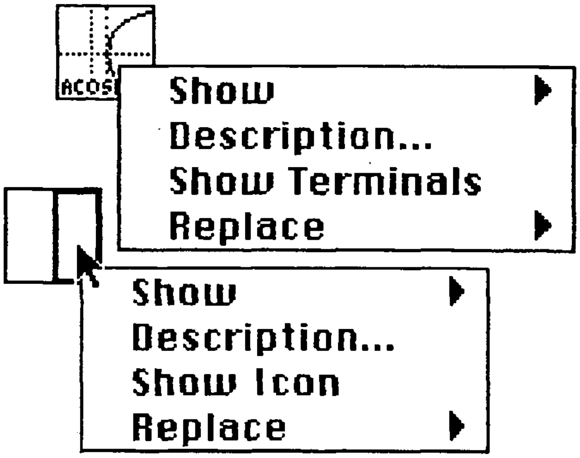
Functions

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. See the *LabVIEW Function Reference Manual* for detailed descriptions of individual functions.

You select functions from the **Functions** menu, as shown below.



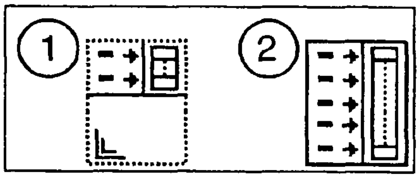
When you select a function, its icon appears on the diagram. You can use the Help window to see how to wire to the function, or you can select **Show Terminals** from the icon pop-up menu to see precisely where the terminals are located, as shown in the following illustration. You can use the function label to annotate its purpose in the diagram.



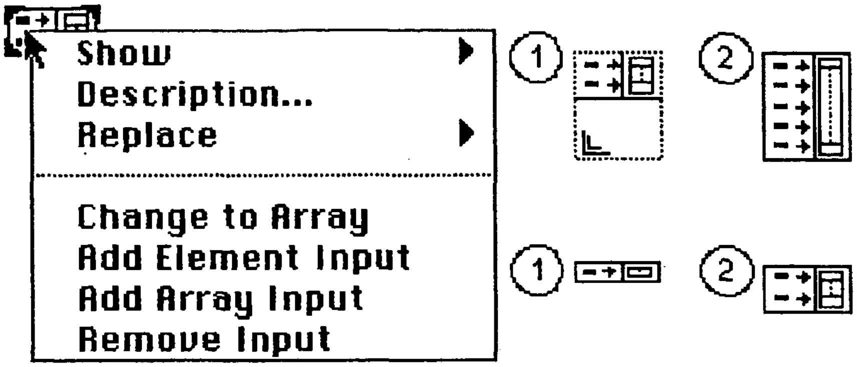
When you wire to a function, you wire to one of its terminals.

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 below. 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 below. 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.



SubVI Nodes

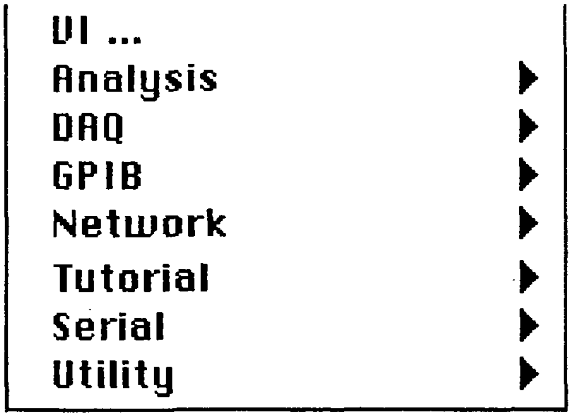
If you design an icon and connector for a VI, you can *call* the VI as a subVI from the diagram of another VI.

A subVI node appears on the block diagram as the icon/connector of the underlying VI. These are the same symbols that you create for the icon/connector pane of the VI. You can show either the icon or the connector in the same manner as for functions. The Help window shows how to wire to the subVI node. The input and output terminal names are the labels you gave to the front panel controls and indicators associated with the connector terminals. Unlike a function, however, a subVI node label displays the name of the subVI.

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

SubVIs combine the benefits of subroutines and VIs. Like subroutines, subVIs employ re-usable code and code sharing; they separate tasks and break down complex problems into manageable units. Like VIs, subVIs are intuitive, graphical user interfaces with self-documenting graphical code and interactive execution, execution highlighting, and printing capabilities. You cannot use a VI recursively; that is, it cannot be its own subVI, or a subVI of one of its subVIs, and so on. You can view the hierarchy of VIs loaded into memory by selecting **Show Hierarchy** from the **Windows** menu.

You select VIs you want to use as subVIs from the bottom half of the **Functions** menu, as shown in the following figure.



VI... Displays the **Open File** dialog box, from which you can select any VI in the system.

Analysis Displays a hierarchical menu of analysis VIs, including digital filter, DSP, numerical, and statistical VIs. See your LabVIEW analysis VI reference manual.

DAQ Displays a hierarchical menu of data acquisition VIs, including analog, digital, data acquisition, counter/timer, waveform, and RTSI VIs. See the *LabVIEW for Windows Data Acquisition VI Reference Manual.*

GPIB Displays a submenu of VIs that follow the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987. See the *LabVIEW GPIB and Serial Port VI Reference Manual.*

Network Displays a menu of VIs for networking with LabVIEW, described in the *LabVIEW Networking Reference Manual.*

Serial Displays a menu of VIs for serial port control. The *LabVIEW GPIB and Serial Port VI Reference Manual* describes these VIs.

Tutorial Displays a menu of VIs that correspond to the exercises in the *LabVIEW for Windows Tutorial.*

Utility Displays a menu of specialized VIs described in Chapter 17, *File VIs*, and in Chapter 18, *Error Handler VIs*, of this manual.

You open a VI by double-clicking on the subVI node or by choosing the **Open Front Panel** command from the node pop-up menu. You can also select the VI from either the **This VI's Callees** or the **Unopened SubVIs** palette in the **Windows** menu. The first palette identifies all the subVIs directly called by the VI in the first window. The second palette identifies all VIs that are in memory but whose panels are not open.

If you do not wire a subVI input terminal, the subVI uses the default value for the respective control as defined when you created that subVI.

Any changes you make to a subVI alters only the version in memory until you save it. However, the changes affect all calls to the subVI and not just the node you used to open the VI. If you make changes and then close the VI without saving them, a dialog box reminds you that the altered memory version is what executes when you run the calling VI. (You can choose **Revert** from the **File** menu to discard changes in memory before you close the front panel of the subVI.) Eventually, when you close the calling VI, another dialog box asks if you want to save the altered version of the subVI.

Structures

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 are otherwise 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.

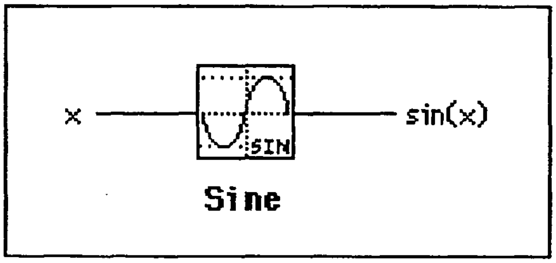
Besides the For Loop, LabVIEW also has a While Loop structure, which repeats the execution of its subdiagram while a condition is TRUE; a Case structure, which has multiple subdiagrams, only one of which executes depending on the value passed to its selector terminal; and a Sequence structure, which 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 Chapter 13, *Structures*, for more information.

Online Help for Constants, Functions, and SubVI Nodes

The LabVIEW Help window, shown in the following figure, displays the wiring diagrams of functions and subVI nodes and the values of universal constants. To display the window, choose **Show Help Window** from the **Windows** menu. If your keyboard has a <Help> key, you can press the <Help> key instead. You can reposition the Help window anywhere on the screen. To close the window, click on the menu box in the upper-left corner of the Help window, or select **Show Help Window** again to uncheck it.

When you move any tool across a function or a subVI node, the Help window shows the icon for the function or VI with wires of the appropriate data type and color attached to each terminal. Input wires point to the left, and output wires point to the right. Terminal names appear with each wire. The following illustration shows the contents of the Help window when you pass a tool over the Sine function.



Help Window

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

The labels of the front panel controls and indicators provide the corresponding terminal names for subVI nodes. All subVI node inputs are optional, and their default values do not appear in the Help window.

The Help window also shows help for VIs in the function menu. Hold the cursor on a VI palette icon for several seconds to see the associated help screen.

Hold a tool over a wire to see the data type carried by that wire.

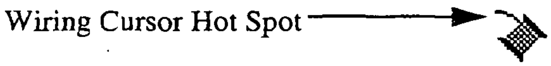
Chapter 12

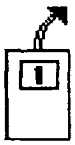
Wiring the Block Diagram

This chapter explains how to wire the block diagram, which is the VI program, and how to debug nonexecutable VIs.

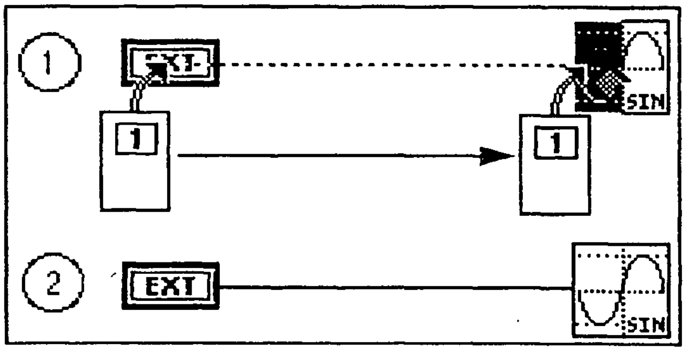
Basic Wiring Techniques

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 below.





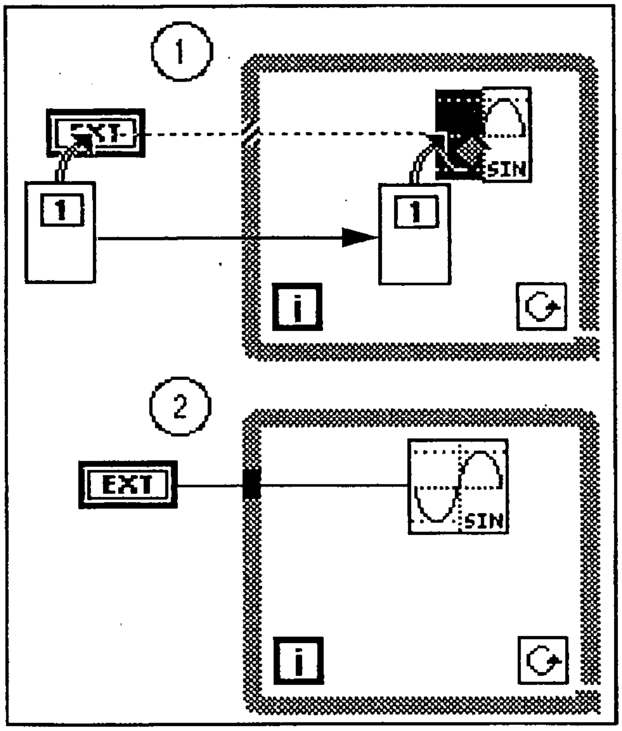
The symbol to the left represents the mouse. In the wiring illustrations in this chapter, the arrow at the end of this mouse symbol shows where to click, and the number printed on the mouse button indicates how many times to click.



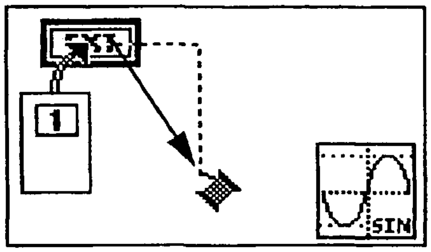
To wire from one terminal to another, click the Wiring tool on the first terminal, move the tool to the second terminal, and then click on the second terminal as shown in the above illustration. It does not matter which terminal you click on first. The terminal area blinks when the hot spot of the Wiring 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 described above, 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.

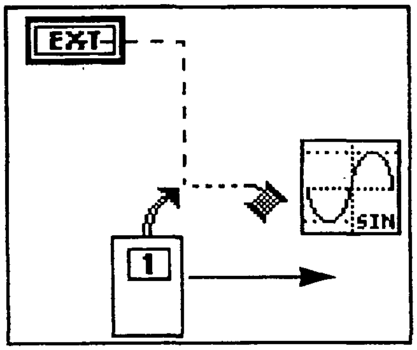
You can wire directly from a terminal outside a structure to a terminal within the structure using the basic wiring operation. LabVIEW creates a *tunnel* where the wire crosses the structure boundary.



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 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.

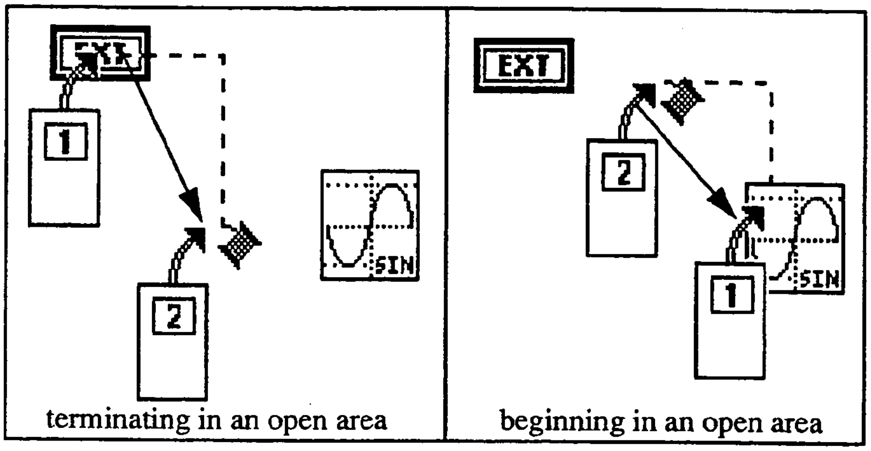


You can elbow your wire once without clicking. You click the mouse to tack the wire and change direction, as shown below.

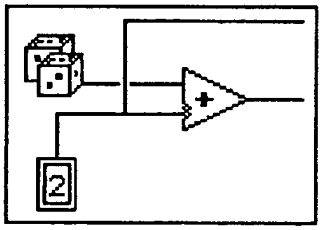


Note: ***You can also change the direction from which the wire leaves a tack point by pressing the space bar. You can <control>-click to untack the last tack point. If the last tack point is the terminal, untacking removes the wire completely.***

You can double-click with the wiring tool to begin or terminate a wire in an open area.

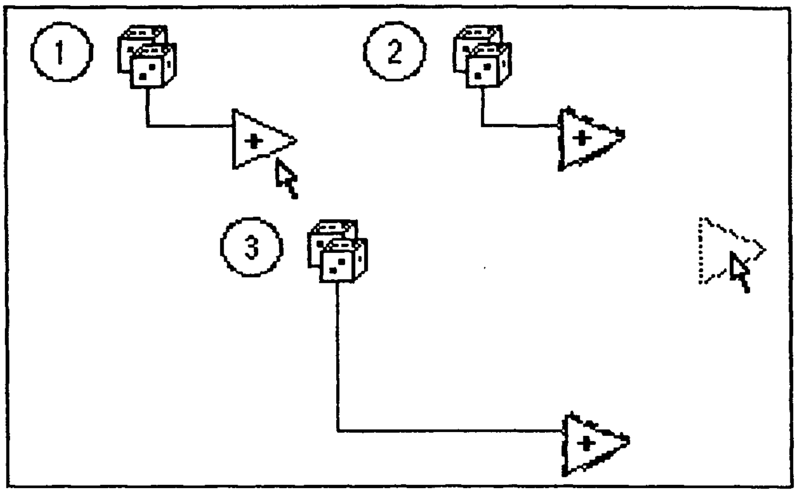


When wires cross, a small gap appears in the first wire drawn, as if it were underneath the second wire, as shown below.



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.

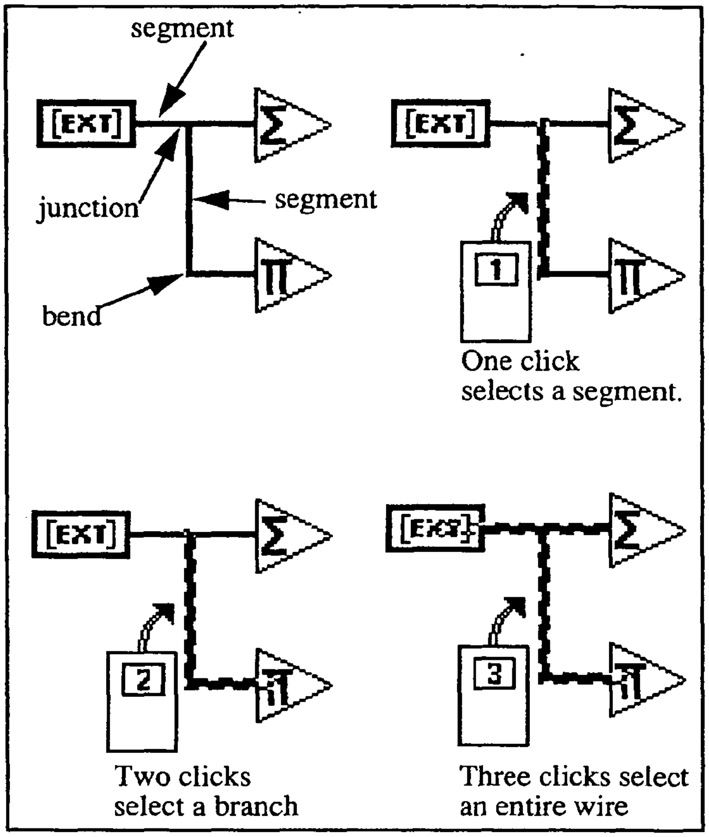


If you duplicate the selected objects or move them from one diagram into another–for example, from the block diagram into a structure diagram–LabVIEW leaves behind the connecting wires, unless you select them as well.

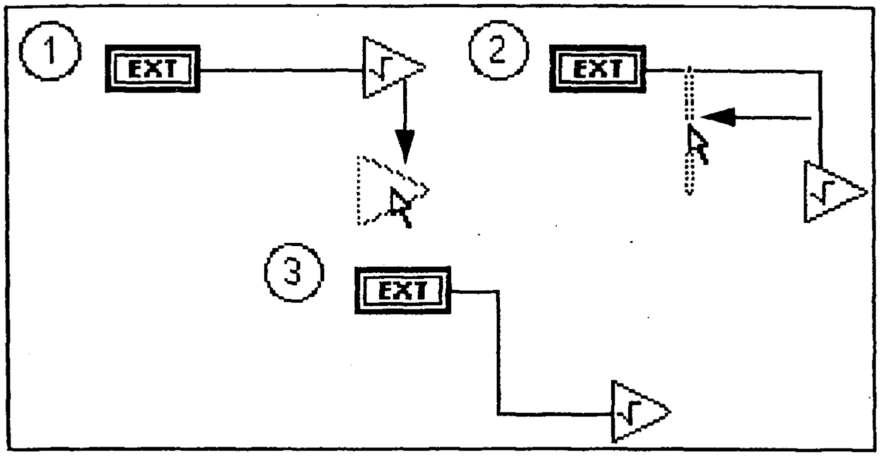
Wire stretching occasionally creates wire stubs or loose ends. You must remove these by using the **Remove Bad Wires** command from the **Edit** menu before the VI will execute.

Selecting, Moving, and Deleting Wires

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

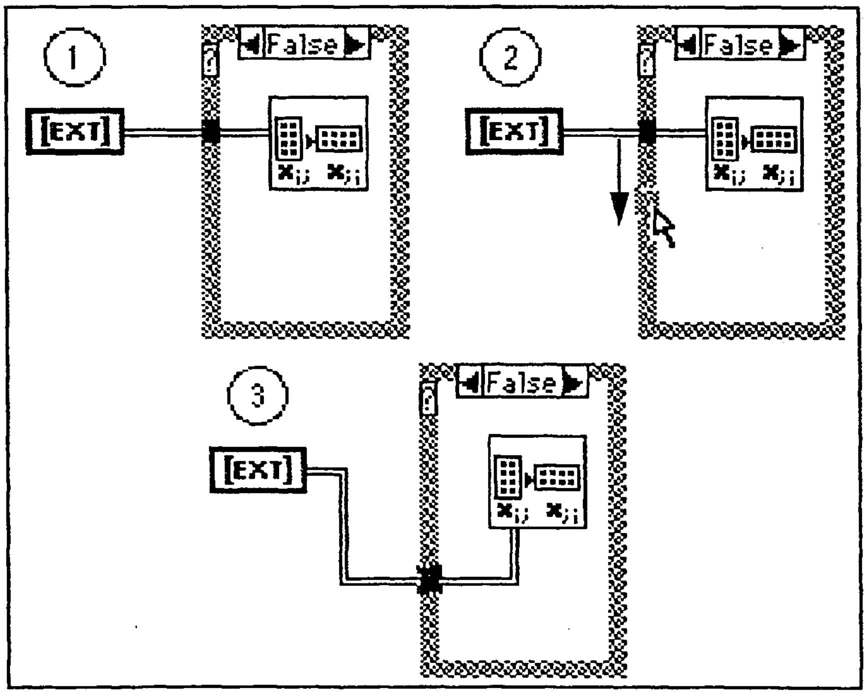


To reposition a wire segment, drag it to the new location with the Positioning tool. You can reposition one or more segments by selecting and dragging them. You can also move selected segments one pixel at a time by pressing the arrow keys on the keyboard. LabVIEW stretches adjacent, unselected segments to accommodate the change. You can select and drag multiple wire segments, even discontinuous segments, simultaneously.

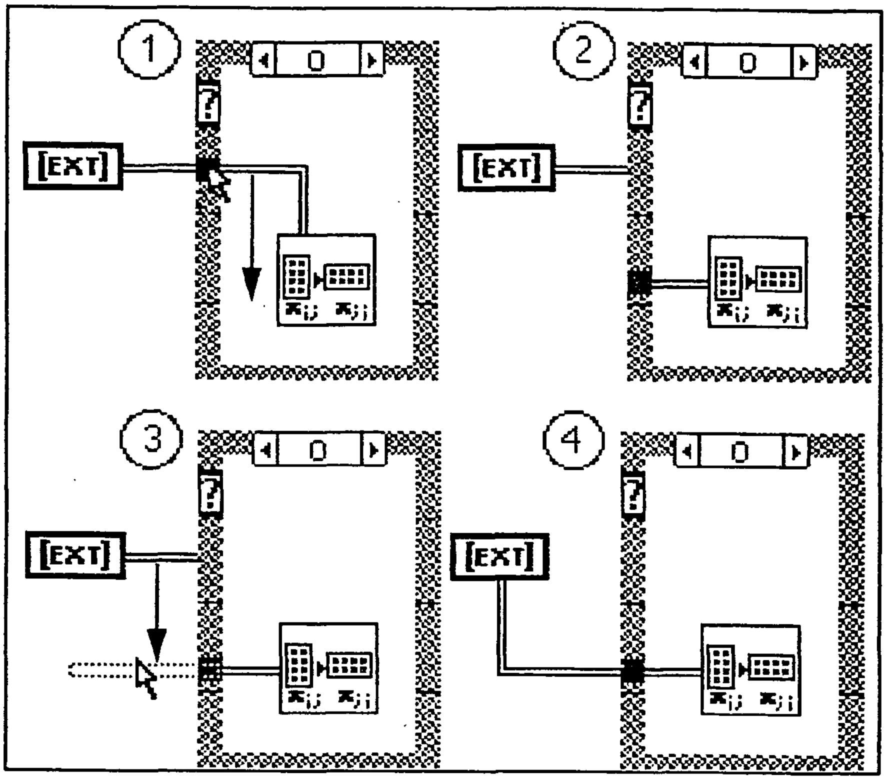


Sometimes repositioning a wired object results in an extra segment or oddly positioned wire segment, as shown in the example above. If you do not like the way the wire looks, you can move it. You can use the shift key to constrain the wire drag.

When you move a tunnel, LabVIEW normally maintains a wire connection between the tunnel and the wired node, as shown below.



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 below. 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. Holding down the shift key allows you to add an object to a group of previously selected objects. 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.

Dragging the Wiring tool slightly past the edge of the Diagram window while you are wiring automatically scrolls the diagram.

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. You can duplicate all objects except for front panel control and indicator terminals. To duplicate these terminals, you must move the corresponding front panel controls and indicators to the new front panel and then rewire the terminals.

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 or by <shift>-clicking on it, 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** menu and rewire. You can also use the **Replace** option in the Increment node pop-up menu. Selecting **Replace** gives you the **Functions** menu, 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.

Wire pop-up menus have an **Insert** option. Choosing **Insert** accesses the **Functions** menu, 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.

Solving Wiring Problems

When you make a wiring mistake, a *broken wire*–a dashed line–appears instead of the wire pattern. The following sections tell you how to recognize, avoid, and solve wiring problems.

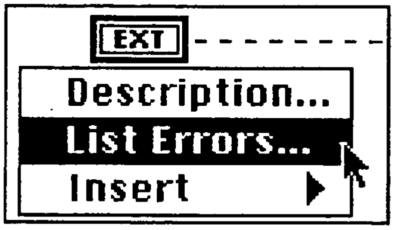
Common Reasons for Bad Wires

This section describes some of the common causes of faulty wires.

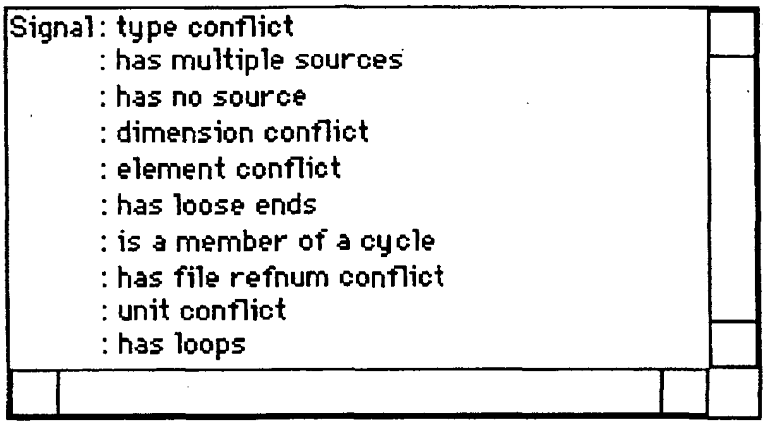
Faulty Connections

If a wire connection is faulty, the wire is broken. If you already know why a wire is broken, choose **Remove Bad Wires** from the **Edit** menu 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 arrow is broken 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 arrow returns to its unbroken state, you corrected the problem. If not, click on the broken Run Arrow 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 below appears and lists the errors. Click on one of the errors. This selects the erroneous wire, which you can then delete or repair.



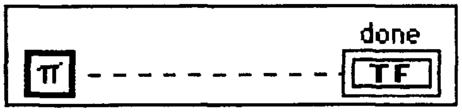
The following figure shows a list of all possible wire errors. They cannot all occur simultaneously for a single wire.



```
Signal: type conflict
      : has multiple sources
      : has no source
      : dimension conflict
      : element conflict
      : has loose ends
      : is a member of a cycle
      : has file refnum conflict
      : unit conflict
      : has loops
```

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.

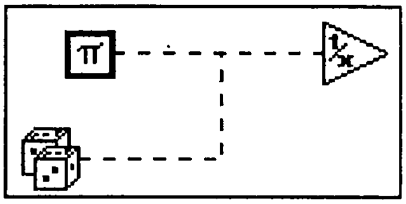


The dimension conflict and element conflict errors occur in similar situations: when you wire two arrays together whose elements match but whose dimensions do not, and when you wire two clusters whose elements have type differences, respectively. The unit conflict occurs when you wire two objects together that have incommensurable units.

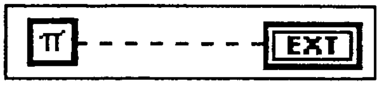
These problems typically arise when you inadvertently connect a wire to the wrong terminal of a function or subVI. Select and remove the wire and rewire to the correct terminal. You can use the Help window to avoid this type of error. In other situations, you may have to change the type of one of the 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 below you must disconnect one source.

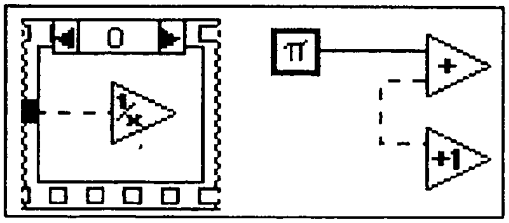


During front panel construction, you may have dropped a control when you meant to drop an indicator. If you try to wire an output value to the terminal of a front panel control, you get a multiple sources error. Pop up on the terminal and select the **Change To Indicator** command to correct this error.



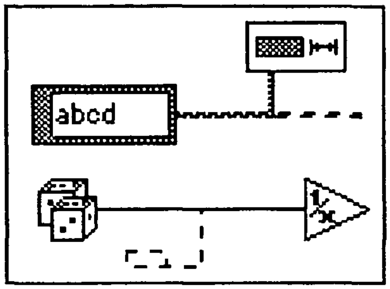
No Wire Source

Below are two examples of wires with no sources. 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.



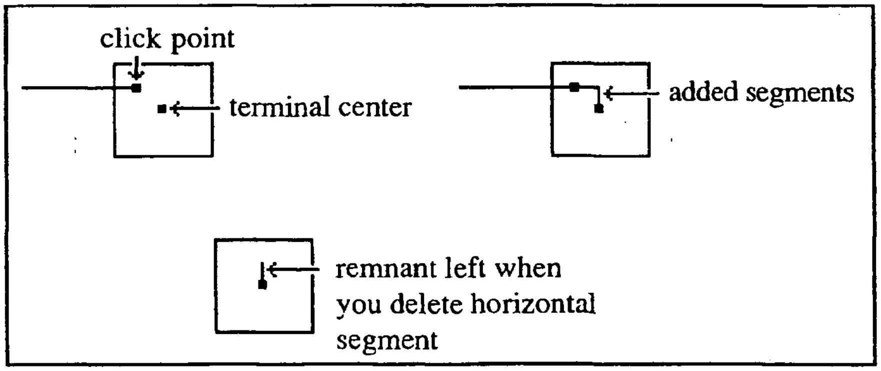
Loose Ends

Loose ends 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

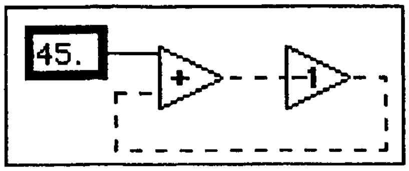
When you wire to a terminal, you seldom click when the cursor is exactly at the center of the terminal. When you are off center, *LabVIEW automatically adds a wire segment from the click point to the terminal center*. If you then remove the wire going to the terminal, a wire stub can remain, causing a broken Run button.



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. LabVIEW cannot execute cycles because each node waits on the other to supply it data before it executes.



The shift register section of Chapter 13, *Structures*, describes the correct way to feed back data in a repetitive calculation.

File Refnum Type Conflict

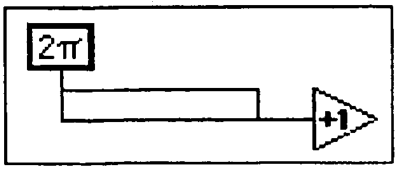
You get a file refnum type conflict when you wire a refnum for a datalog file to another datalog file that has a different record type.

Wiring Situations to Avoid

The following sections 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. 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

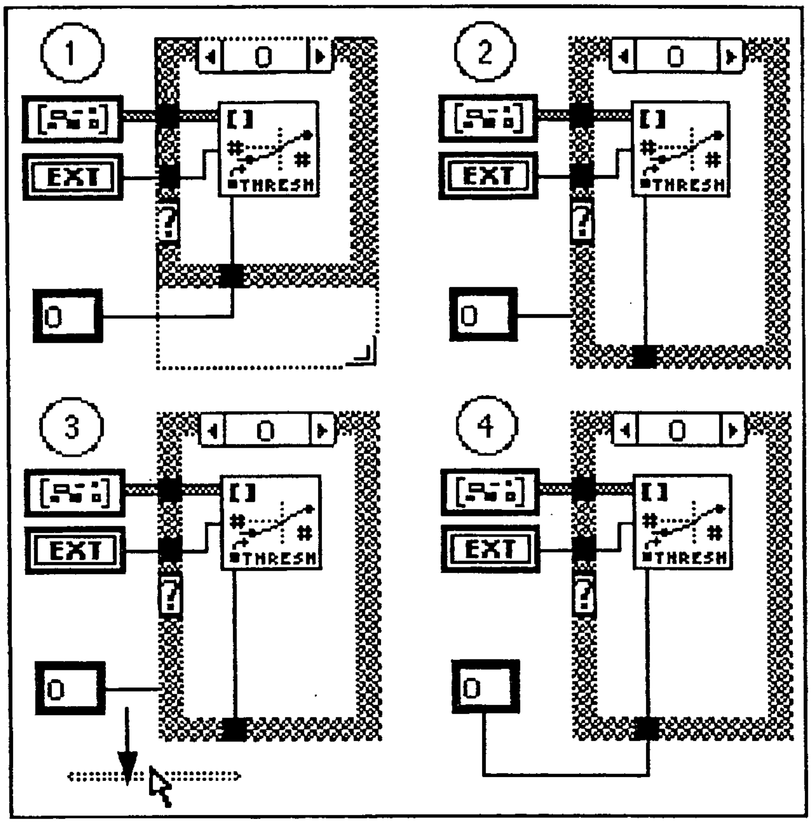
A loop of wire is not an error but is poor design because it unnecessarily clutters the diagram. Double-click on one of the branches to select it, then delete it.



Note: ***If your VI has a loop of wire, a warning will appear in the error list box with the message*** `has loops`***, but only if there is also some other error making the VI nonexecutable.***

Hidden Wire Segments

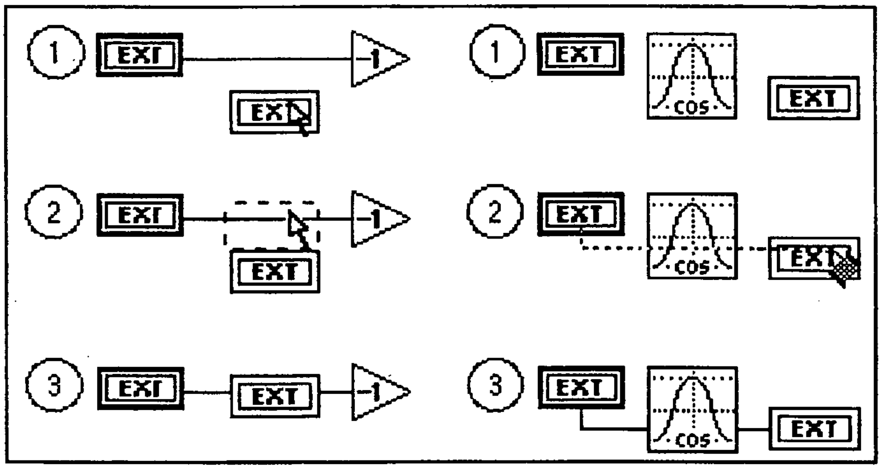
Try to avoid wiring under a structure border–wire through a tunnel to pass data into the structure instead. You should also avoid wiring between overlapped objects.



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 example above. 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 in the illustration on the left below. Dragging a wire through an icon or terminal also appears to make a connection, but the wire is actually behind the icon, as shown in the illustration on the right below. Avoid these situations because they are visually confusing.



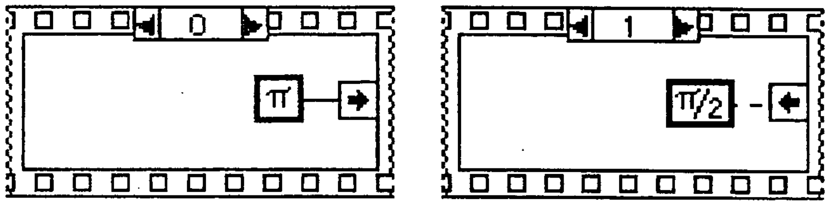
See the *Warnings for Overlapping Objects* section in Chapter 3 for more information on this problem.

Problems in Wiring Structures

The following sections discuss faulty connections with structures.

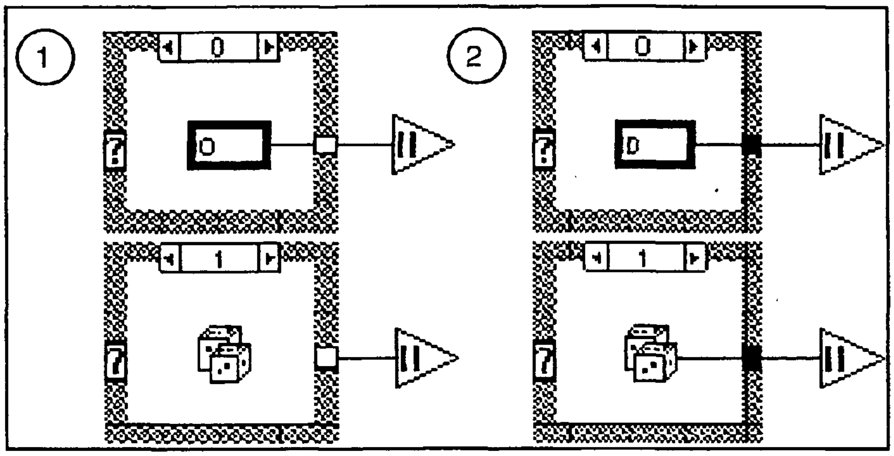
Assigning More Than One Value to a Sequence Local

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.



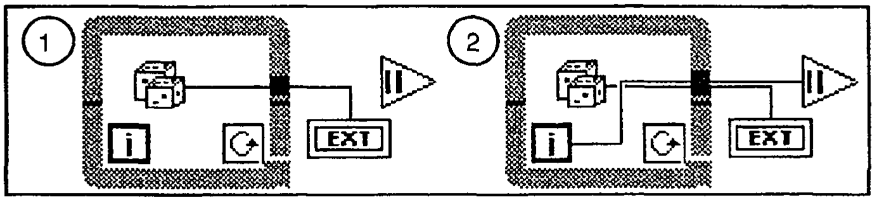
Failing to Wire a Tunnel in All Cases of a Case Structure

Wiring from a Case structure to an object outside the structure results in a bad tunnel if you do not connect 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 each execution of the Case structure.

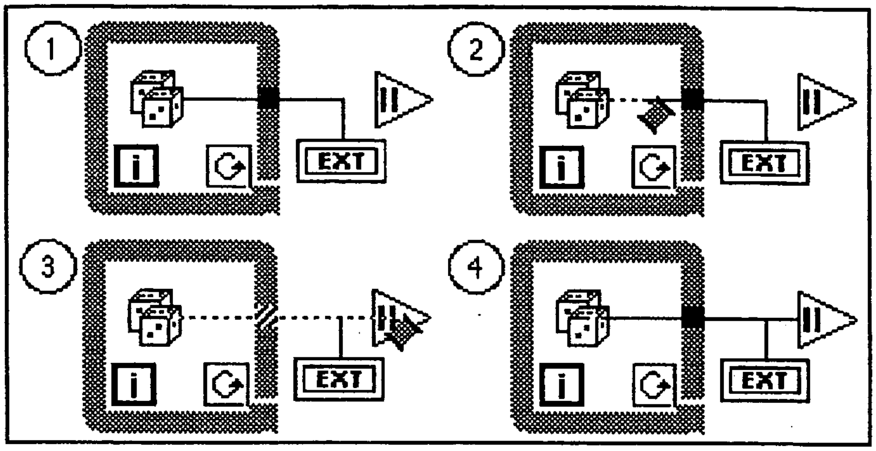


Overlapping Tunnels

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.



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, as shown above, although you can always remove the bad wires and try again.



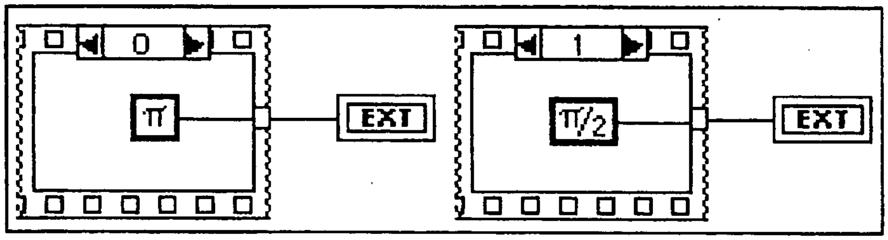
If you need to wire from an object inside a structure to an object outside when one such wire already exists, *do not wire* through the structure again as shown in the above illustration. Instead, begin the second wire at the tunnel. In this example, two overlapping tunnels do not cause a problem. But 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.

See the *Warnings for Overlapping Objects* section in Chapter 3 for more information on this problem.

Note: ***If your VI has overlapping tunnels, a warning will appear in the error list box if there is also some other error making the VI nonexecutable.***

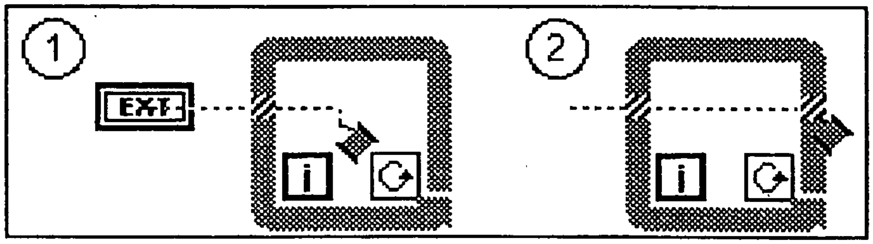
Wiring from Multiple Frames of a Sequence Structure

This 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.

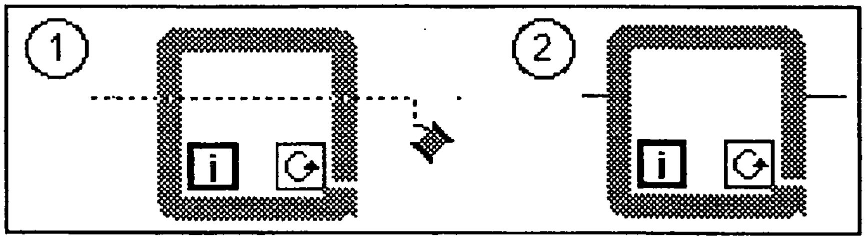


Wiring Underneath Rather Than Through a Structure

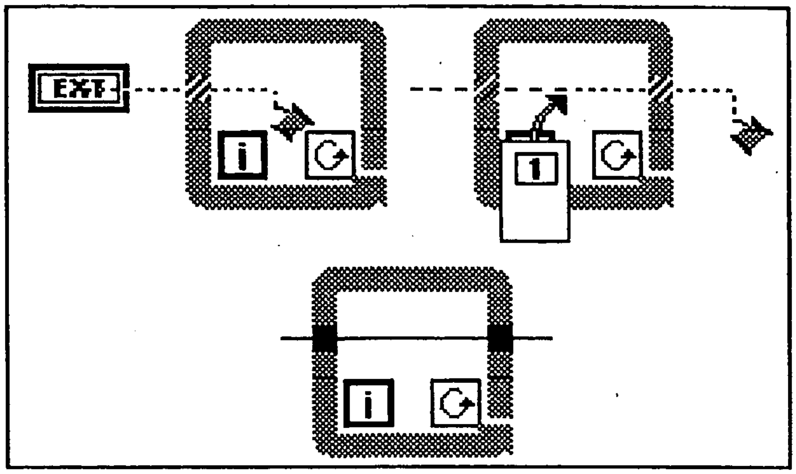
To wire through a structure you must click either in the interior or on the border of the structure, as shown below.



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.



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.

Debugging Techniques for Nonexecutable VIs

Fixing a Broken VI

A VI cannot compile or run if it is broken. The VI is usually 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 **Remove Bad Wires** from the Edit menu first. Often, this fixes a broken VI.

To find out why a VI is broken, click on the broken run button. An information box titled *VI Name* errors appears listing all the errors. To locate a particular error, 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. 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 must remove even wire stubs you cannot see with the **Remove Bad Wires** command from the **Edit** menu.

- 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.

The following list contains some possible errors in a form similar to the way they look in the VI error information box.

- `*Function Name*: has 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.
- `*Control*: control is being edited in a Control Editor window.` You cannot run a VI if one of the controls is being edited in the control editor.
- `Enumeration has duplicate entries.` All the items in an enumeration control must be unique and distinct.
- `Errors: stopped after *N*.` LabVIEW shows only *N* 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 doesn't exist.` The name of a variable has changed since you last loaded the VI and the name in the Global or Local no longer matches. Pop up on it and select another name.
- `Global Variable: subVI is missing.` LabVIEW could not find the global VI when it loaded the calling VI, perhaps

because you changed the name. Replace the bad global variable 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.

- `Object: (partly) hidden.` This warning appears only if other errors are present. The object is hidden and may cause confusion later on.

- `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: Can't 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.

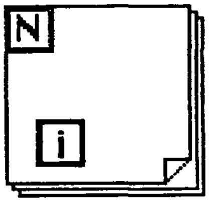
If you encounter messages that are not self-explanatory, and not listed in this chapter, contact National Instruments.

Chapter 13 Structures

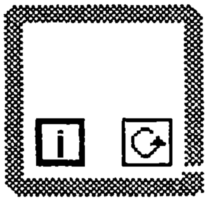
This chapter describes how to use the For Loop, While Loop, Case, and Sequence *structures*, found on the **Structs & Constants** palette of the **Function** menu.

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 below.

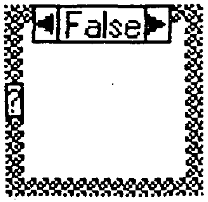
For Loop



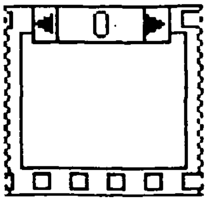
While Loop



Case Structure



Sequence Structure



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 as 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. These terminals are described with the appropriate structures in the following sections.

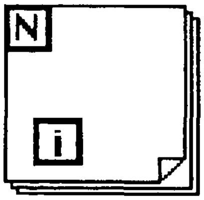
For Loop and While Loop Structures

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

N

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* (see the *Auto-Indexing* section in this chapter for more information). The other edges of the count terminal are exposed to the inside of the loop so that you can access the count internally.

i

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.

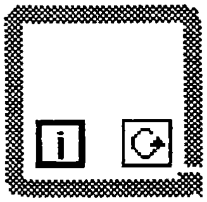
The For Loop is equivalent to the following pseudo-code:

```
for i = 0 to N-1
   Execute subdiagram
```

While Loop

↻

conditional terminal



i

iteration 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.

The iteration terminal behaves exactly as it does in the For Loop.

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

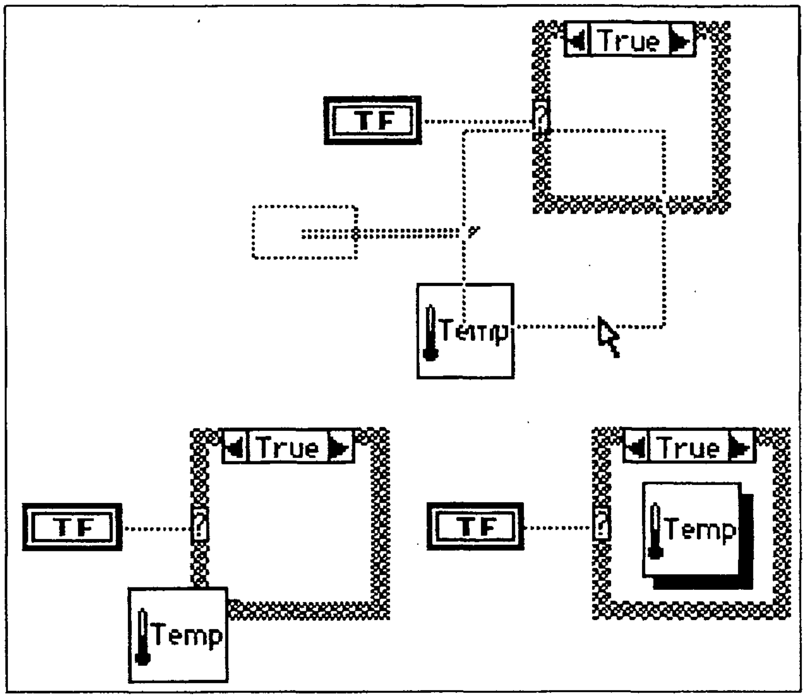
```
Do
   Execute subdiagram (which sets condition)
While condition is TRUE
```

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* section of this chapter for more information.

Placing Objects inside Structures

You can place an object inside structures by dragging it inside, or by creating it in place.

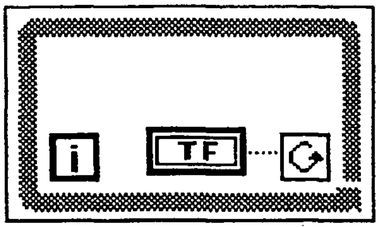
You cannot put an object inside a structure by dragging the structure over the object. If you move a structure and it overlaps another object, the overlapped object will be visible above the edge of the structure. If you put the structure completely over another object, that object will display a thick shadow to warn you that the object is below, not inside the structure. Both of these situations are shown in the following illustration.



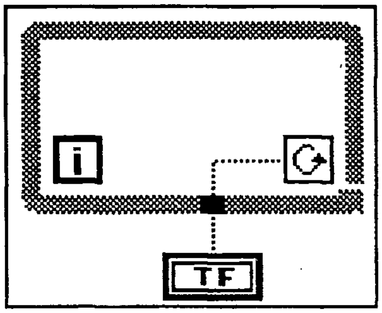
Terminals Inside Loops

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.

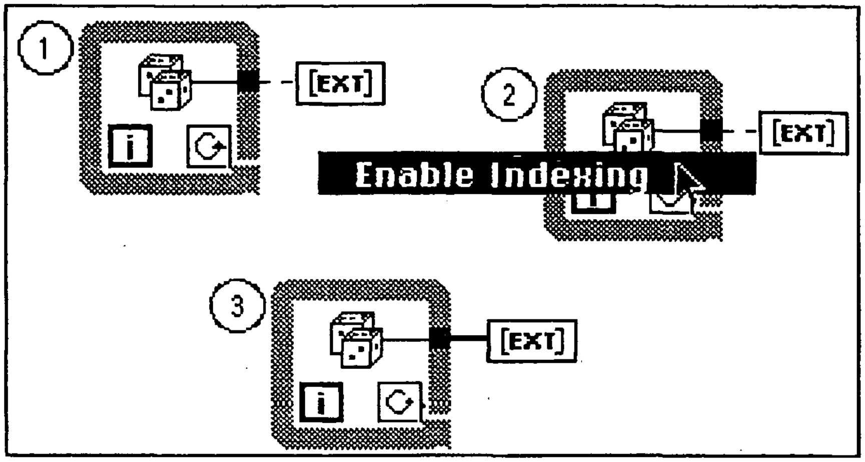


If the control was true at the start, changing the value of the front panel control to FALSE does not stop the execution of this loop because the value is not propagated until the loop stops and the VI is re-run. If you inadvertently create an infinite loop in LabVIEW, you can always stop it by aborting 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 enable auto-indexing and wire an array of any dimension from an external node to an input tunnel on the loop border, components of that array enter the loop one at a time starting with the first component. The loop indexes scalar elements from 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. You must pop up on the tunnel at the loop border and choose **Enable Indexing** for While Loops. You can likewise select **Disable Indexing** from a For Loop tunnel pop-up menu.

Using Auto-Indexing to Set the For Loop Count

When you enable auto-indexing on an array *entering* a For Loop, LabVIEW automatically sets the count to the 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.

Using 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 for Windows Tutorial* ***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.

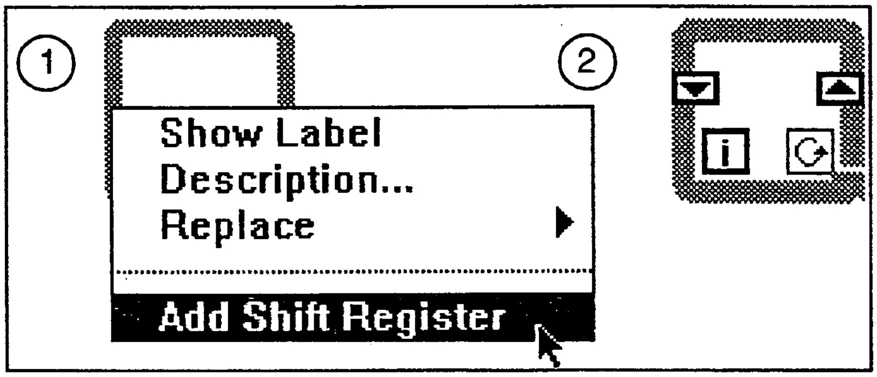
- An output array created by auto-indexing at an output tunnel is empty.
- The output from an initialized shift register is the initial value. See the *Shift Registers* section of this chapter for more information.
- An array passed through a non-indexing output tunnel is also empty.
- All scalars passed 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 if a zero count can occur, and if so, what the effects would be.

Shift Registers

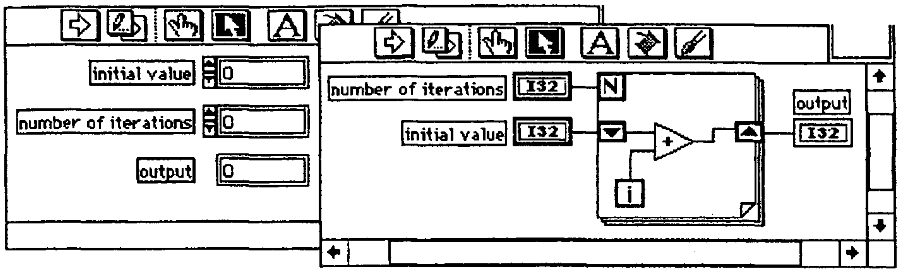
Shift registers, which are available in For Loops and While Loops, are local variables that feed back or transfer values from the completion of one iteration to the beginning of the next. By selecting **Add Shift Register** from the loop border pop-up menu, you can create a register anywhere along the vertical boundary, as shown below. 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.



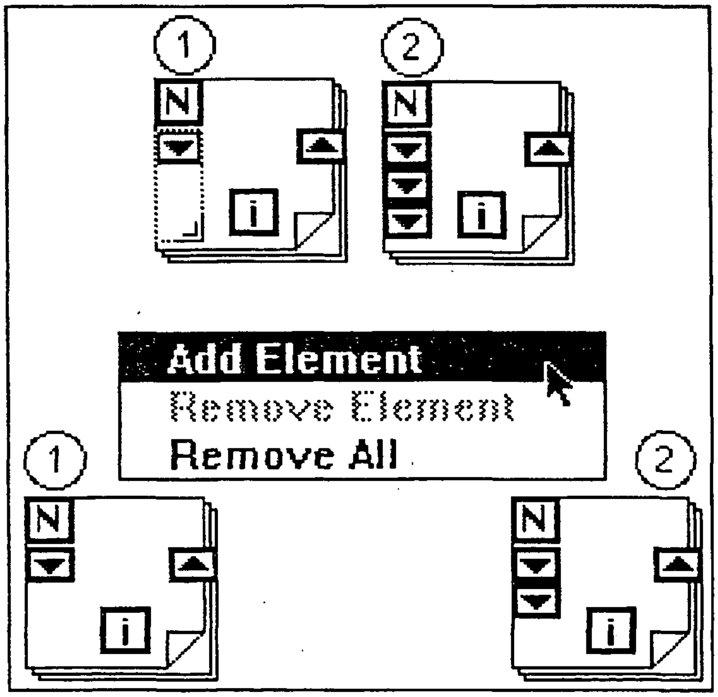
A shift register has a pair of terminals directly opposite each other on the vertical sides of the loop border. The *right terminal,* the rectangle with the up arrow, stores the data at the completion of an iteration. LabVIEW shifts that data at the end of the iteration, and it appears in the *left terminal,* 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.

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. If you initialize one shift register on a structure, you must initialize all the shift registers on that structure. You should normally use initialized shift registers to ensure consistent behavior. See Chapter 3, *Loops and Charts*, of the *LabVIEW for Windows Tutorial* 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.



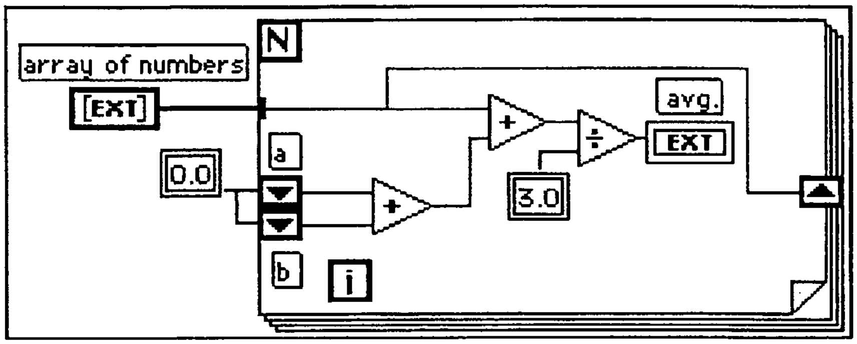
To add or remove terminals, use the Positioning tool to resize the left terminals. Alternatively, you can use the **Add Element** command from the shift register pop-up menu to add more left terminals to the shift register. Added terminals appear directly below the one on which you pop up, as shown in the following illustration. 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 topmost left 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.

The following pseudo-code shows a three-value running average routine equivalent to the LabVIEW block diagram.

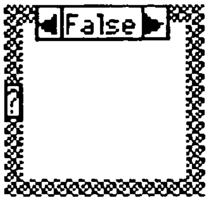
```
a=b=0
for i =0 to N-1
    avg = (x[i]+a+b)/3
    b =a
    a =x[i]
```



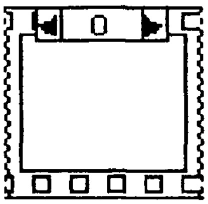
Case and Sequence Structures

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.

Case structure



Sequence structure



Clicking on the decrement (left) or increment (right) button displays the previous or next subdiagram, respectively. Incrementing from the last subdiagram displays the first subdiagram, and decrementing from the first subdiagram displays the last. Other uses of the display window are explained in the *Editing Case and Sequence Structures* section of this chapter.

Case Structure

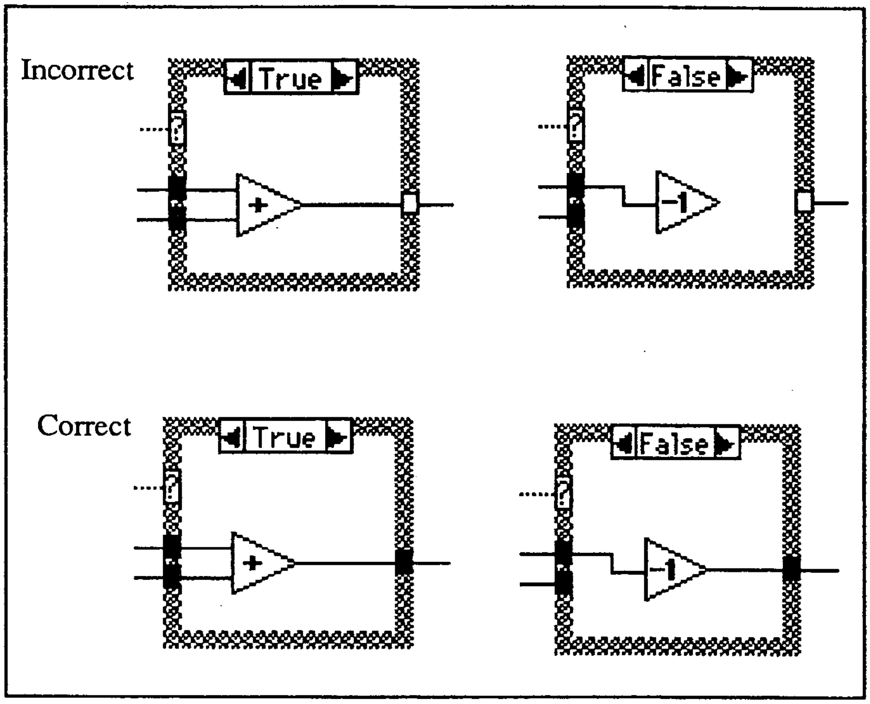
selector

The Case structure has two or more subdiagrams, or *cases*, exactly one of which executes when the structure executes depending upon 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 has two cases, False and True. If a numeric is wired to thc selector, the structure can have from 0 to 2^{15}-1 cases. Initially only the 0 and 1 cases are available. To add more cases, follow the procedure outlined in the *Adding Subdiagrams* section of this chapter. (If you wire a floating-point number to the selector, LabVIEW rounds it to the nearest integer value. LabVIEW coerces negative numbers to 0 and values higher than the highest-numbered case to the last case.)

If you wire an enumeration to the selector, there must be one subdiagram for each enumeration item. The case identifiers will be the same as the enumeration item names.

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 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 above illustration, and the broken Run button appears.

When all cases supply data to the tunnel, it turns black, as in the bottom example in the above 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.

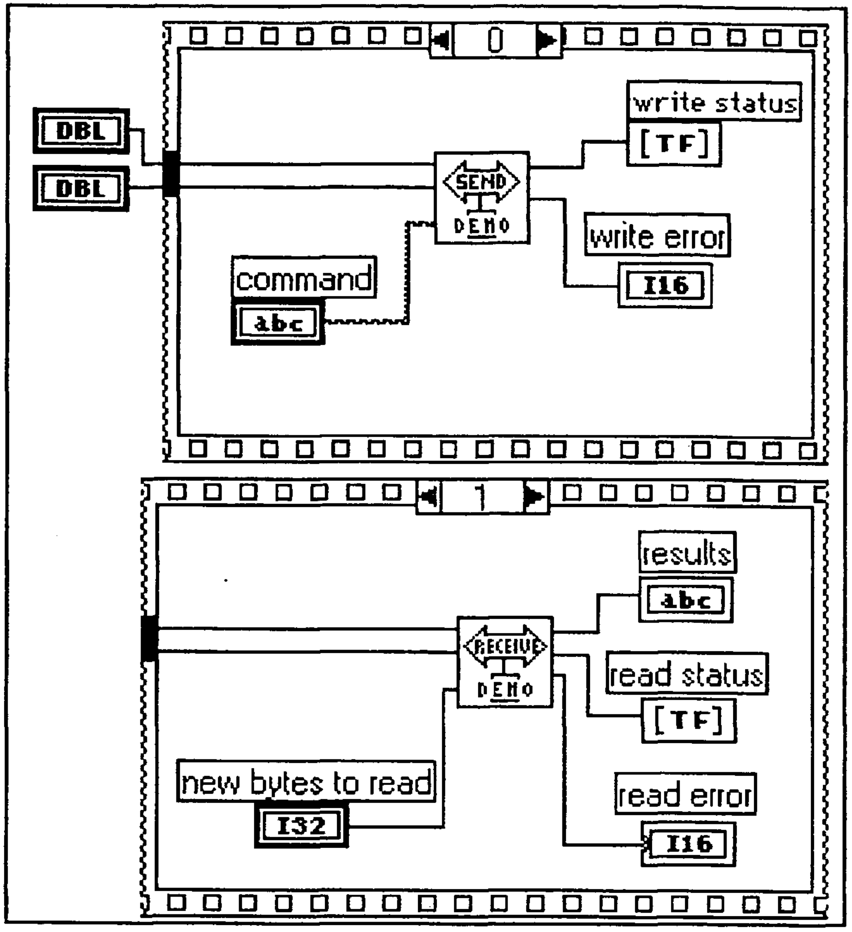
Determining the execution order of a program by arranging its elements in sequence is called *control flow*. BASIC, C, and most other programming languages have inherent control flow, because statements execute in the order in which they appear in the program. The Sequence structure is the LabVIEW way 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 will always execute 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

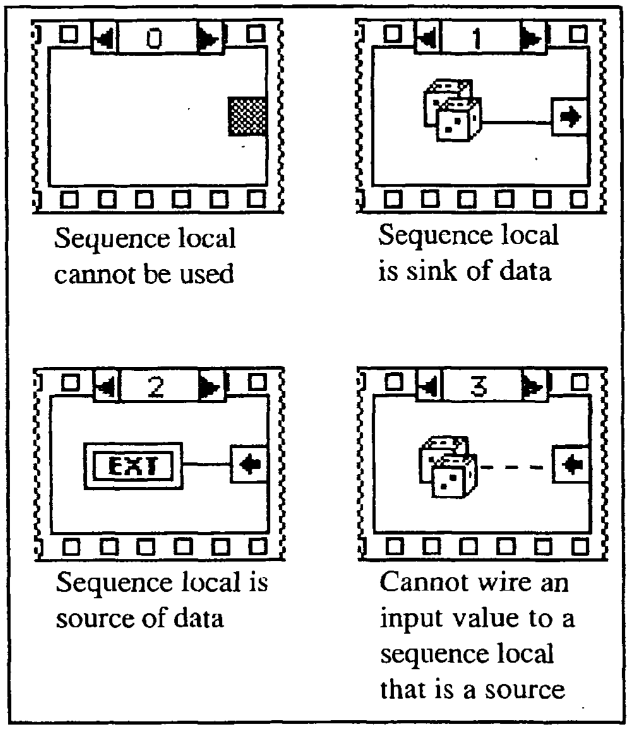
Receive VI in frame 1, as shown in the following illustration, you can enforce the proper execution order.



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 too close to another 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.



Sequence local cannot be used

Sequence local is sink of data

Sequence local is source of data

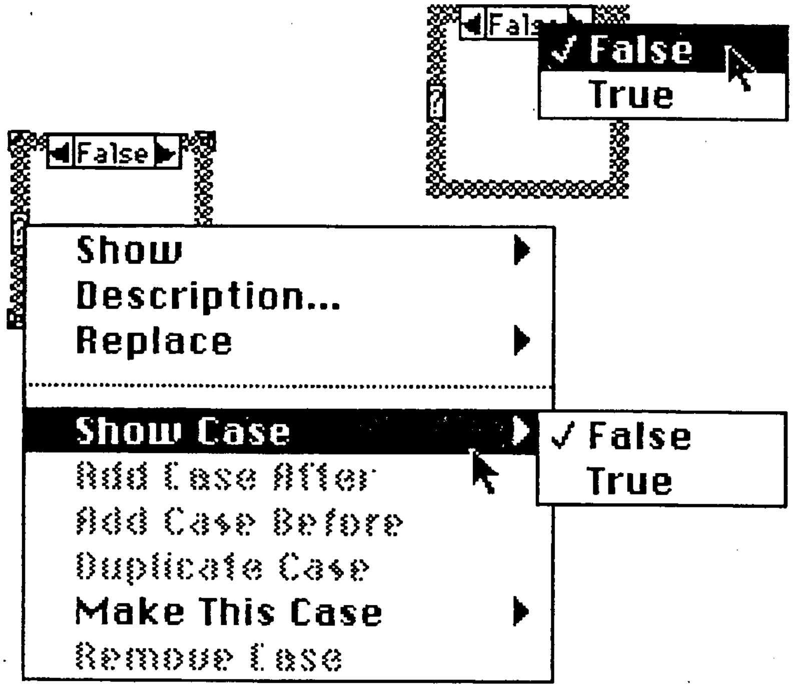
Cannot wire an input value to a sequence local that is a source

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 and can never have fewer. A new Sequence structure has one frame.

Moving between Subdiagrams

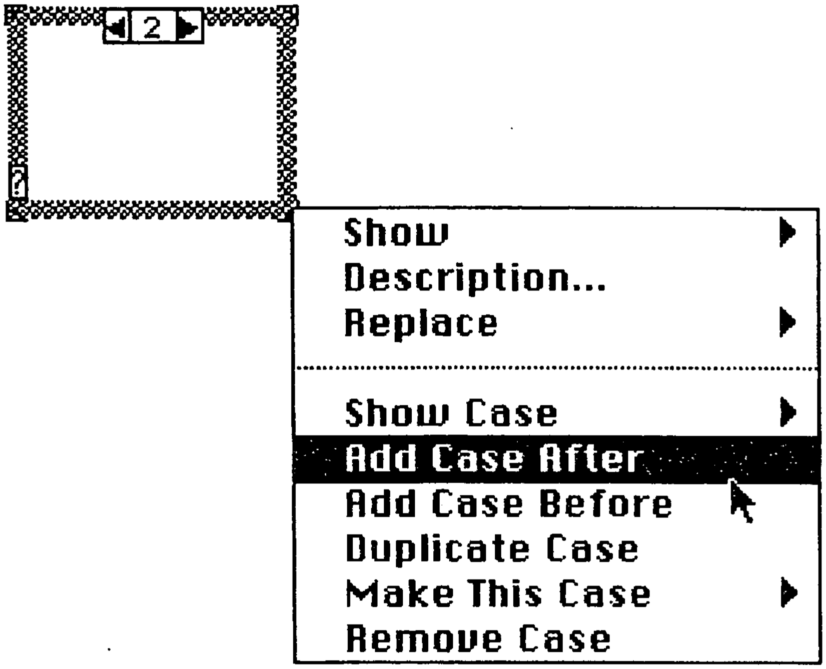
The fastest way to view the next lower or higher subdiagram is to click on the increment or decrement button in the display window. If you want to jump over several subdiagrams, click on the subdiagram identifier and select the destination subdiagram from the pop-up menu, as shown below. You can also use the **Show Case** command from the border pop-up menu.



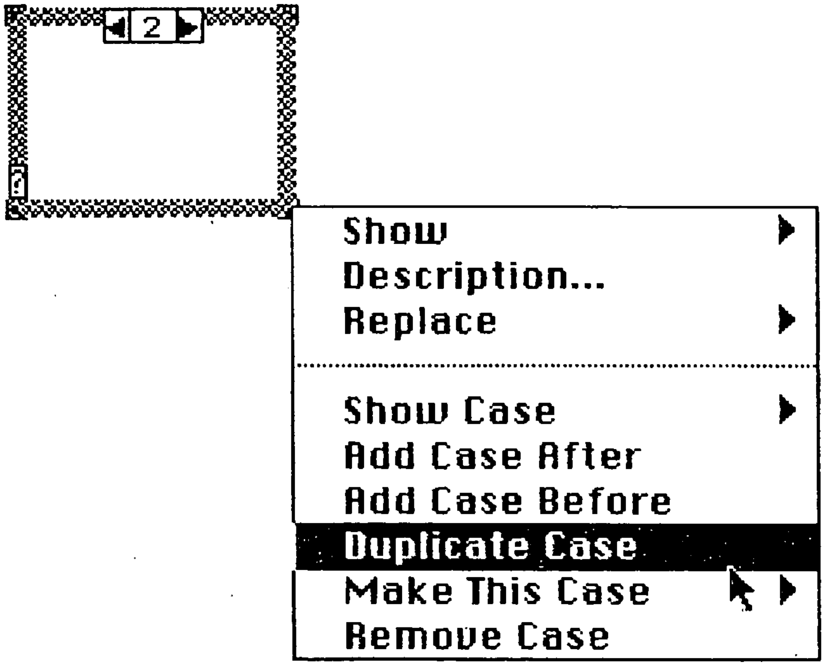
Adding Subdiagrams

You can add subdiagrams several ways.

If you select the **Add Case After** command from the structure border pop-up menu, as shown below, LabVIEW inserts an empty subdiagram after the currently visible one. For example, an empty subdiagram at position 3 will be 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.



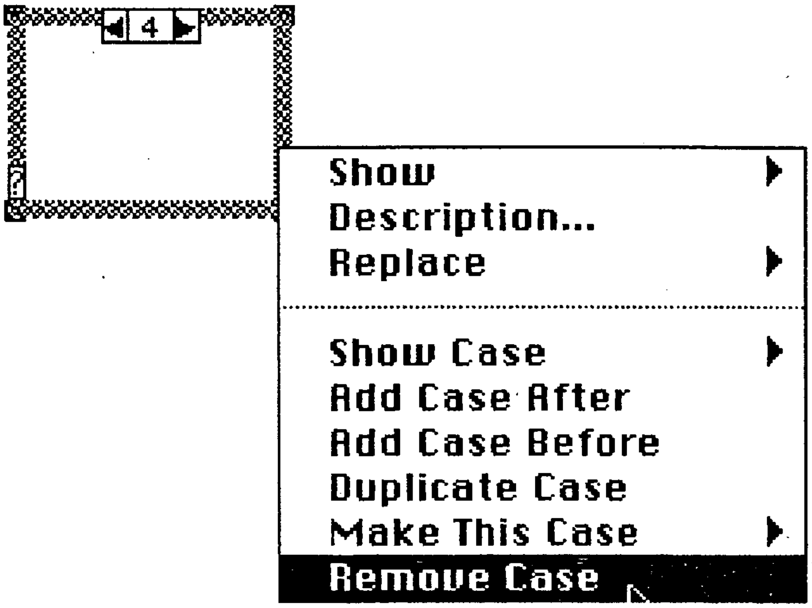
If you select **Duplicate Case** from either the structure border pop-up menu or the diagram identifier pop-up menu, as shown below, a copy of the visible subdiagram is inserted after itself.



These options are not available for a Case structure with a Boolean selector. When you add or remove subdiagrams, LabVIEW automatically adjusts the numeric identifiers to accommodate the inserted or deleted subdiagrams.

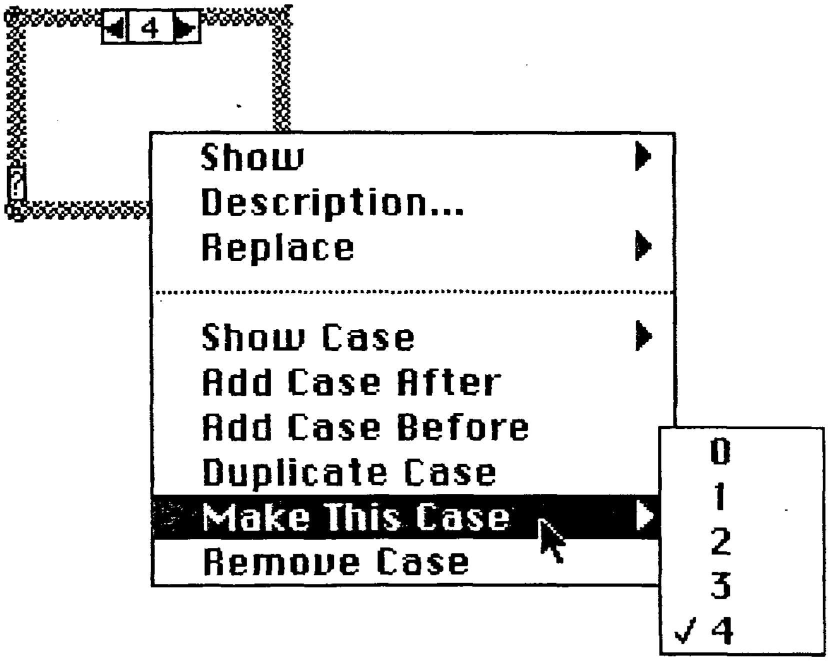
Deleting Subdiagrams

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 two cases–or one frame–exist.



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 values of other subdiagrams.

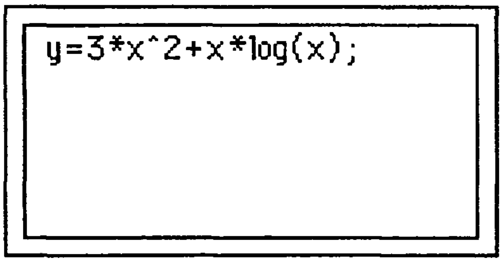


Chapter 14

The Formula Node

This chapter describes how to use the Formula Node to execute mathematical formulas on the block diagram.

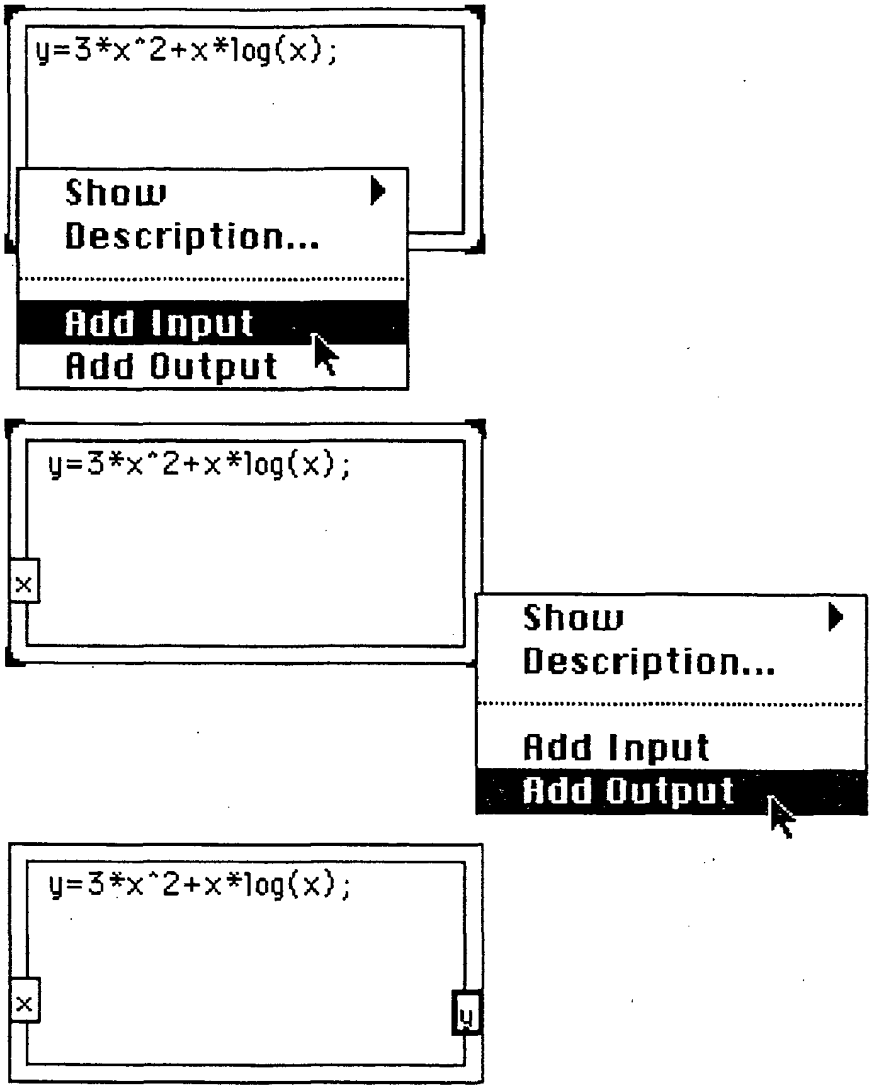
The Formula Node is available from the **Structs & Constants** palette of the **Functions** menu. It is a resizable box similar to the four structures, but instead of containing a subdiagram, the Formula Node contains one or more formula statements delimited by a semicolon, as in the following example.



```
y=3*x^2+x*log(x);
```

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*`*/`).

The pop-up menu on the border of the Formula Node contains options to add input and output variables, as shown in the illustration below.

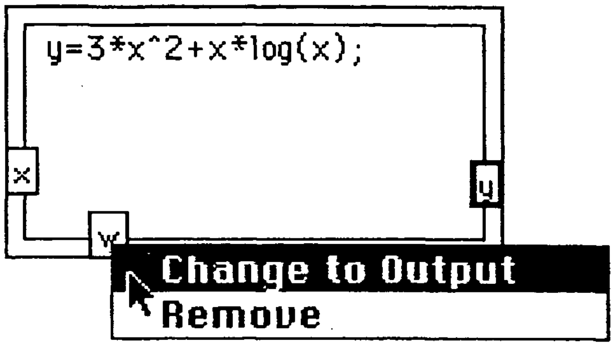


```
y=3*x^2+x*log(x);
```

Output variables are distinguished by a thicker border.

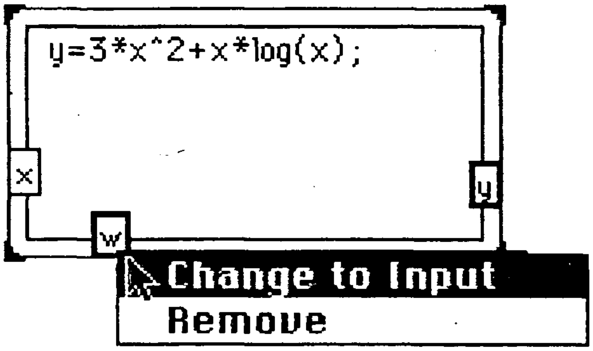
There is no limit to the number of variables or formulas in a Formula Node. No two inputs and no two outputs can have the same name. However, an output can have the same name as an input.

You can change an input to an output by selecting **Change to Output** from the pop-up menu, as shown in the following illustration.



```
y=3*x^2+x*log(x);
```

You can change an output to an input by selecting **Change to Input** from the pop-up menu, as shown below.



```
y=3*x^2+x*log(x);
```

All variables are floating-point numeric scalars, whose precision depends on the configuration of your computer. 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 Functions

All function names must be lower case. Table 14-1 shows the names of the Formula Node functions.

Table 14-1. Formula Node Functions

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,y)	Inverse Tangent	Computes the inverse tangent of y/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 ≥ 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)).

Table 14-1. Formula Node Functions (Continued)

Function	Corresponding LabVIEW Function Name	Description
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 (e^x-1).
floor(x)	Round to -Infinity	Truncates x to the next lower integer (largest int ≤ x).
getexp(x)	mantissa and exponent	Returns the exponent.
getman(x)	mantissa and exponent	Returns the mantissa.
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).
lnpl(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)	maximum and minimum	Compares x and y and returns the larger value.
min(x,y)	maximum and minimum	Compares x and y and returns the smaller value
mod(x,y)	quotient and 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.

Table 14-1. Formula Node Functions (Continued)

Function	**Corresponding LabVIEW Function Name**	**Description**
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	x^y	Computes the value of x raised to the y power.

The Formula Node syntax is summarized below using Bakus-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 non-zero 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. See the Formula Node illustration at the beginning of this chapter for an example of how to implement this syntax.

Table 14-2 lists errors detected by the Formula Node.

Table 14-2. Formula Node Errors

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.

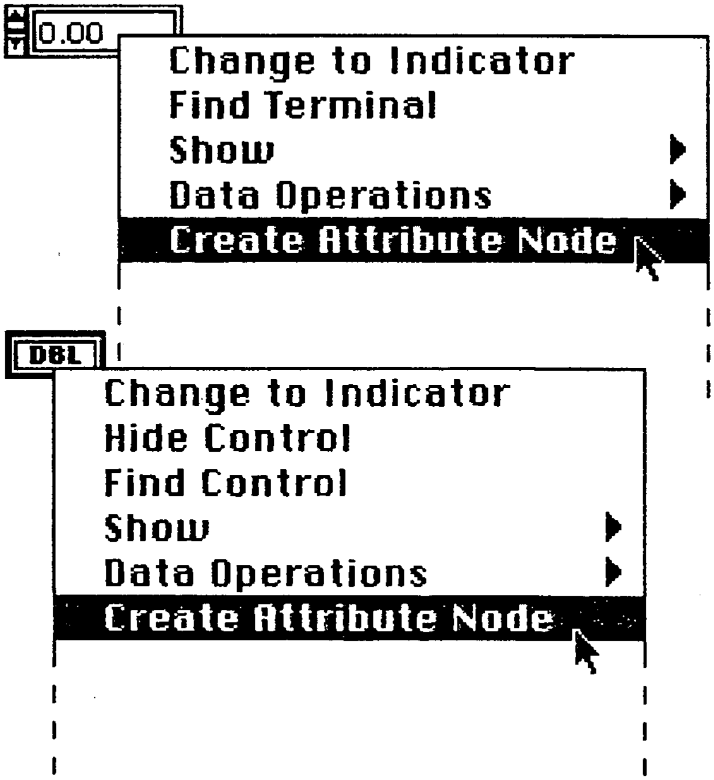
Chapter 15

Attribute Nodes

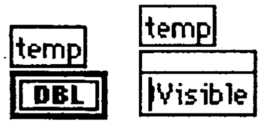
This chapter describes how to use Attribute Nodes to set and read attributes of front panel controls programmatically. Some useful attributes include the display colors, control visibility, menu strings for a ring control, graph or chart scales, and graph cursors.

Creating Attribute Nodes

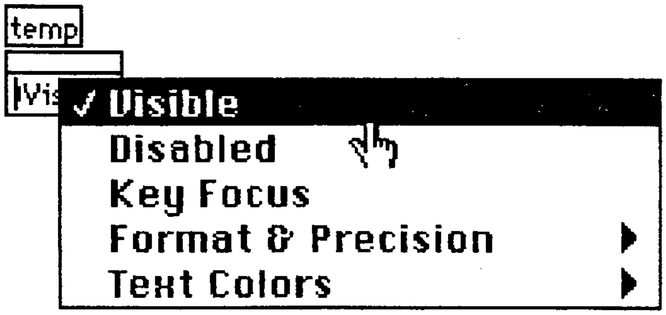
You create an Attribute Node by selecting the **Create Attribute Node** option from the popup menu of a front panel control or the control's terminal. Selecting this option creates a new node on the diagram located near the terminal for the control, as shown in the following illustration.



If the control has a label associated with it, the control's label is used for the initial label of the Attribute Node. You can change the label after the node has been created.

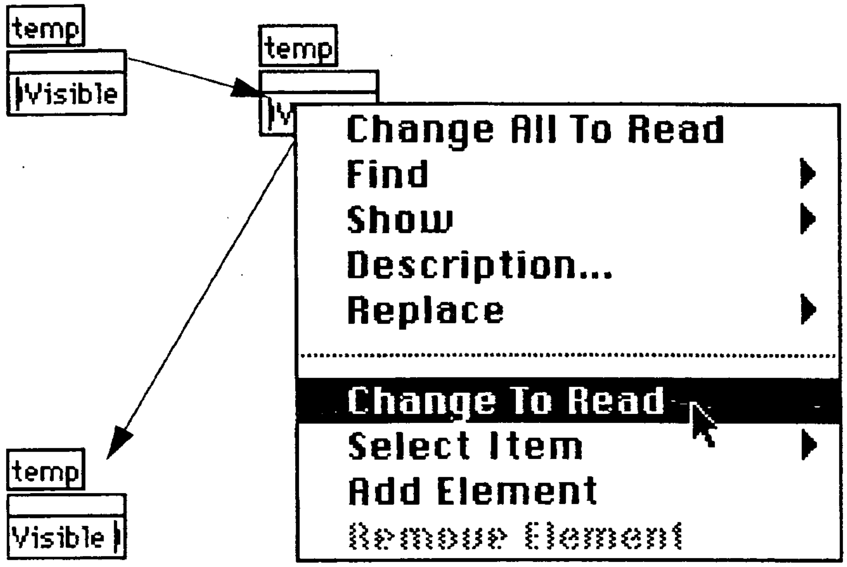


If you click with the operating tool on an attribute terminal, you get a menu of options which you can set for or read from the control, as shown in the following illustration.

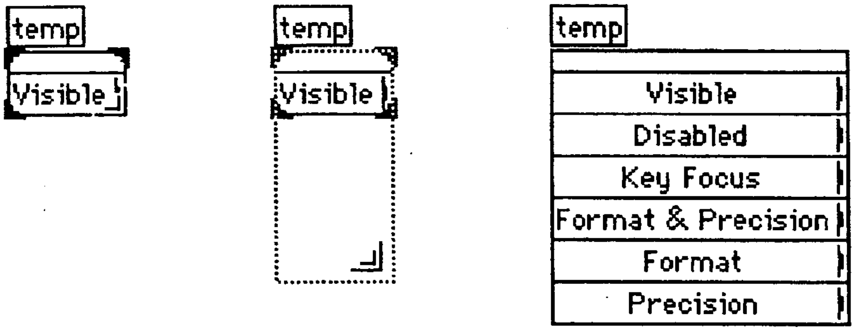


You choose whether to read or set attributes by selecting either the **Change to Read** or **Change to Write** option from the Attribute Node popup menu, as shown in the illustration below. You can set an attribute when the small direction arrow is located on the left side of the terminal.

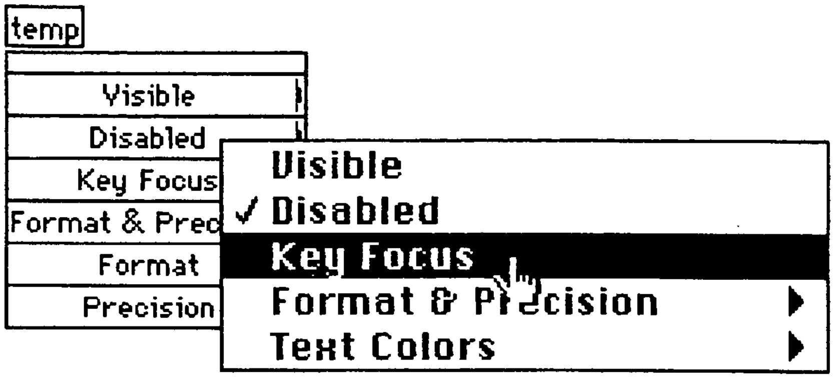
You can read an attribute when the terminal symbol is located on the right side of the terminal.

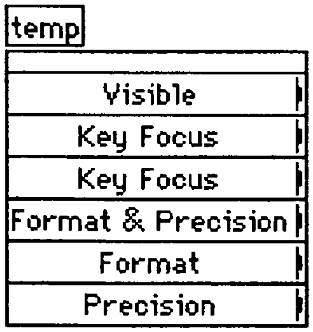


You can read or set more than one attribute with the same node by enlarging the Attribute Node. As it is enlarged, new terminals are added.

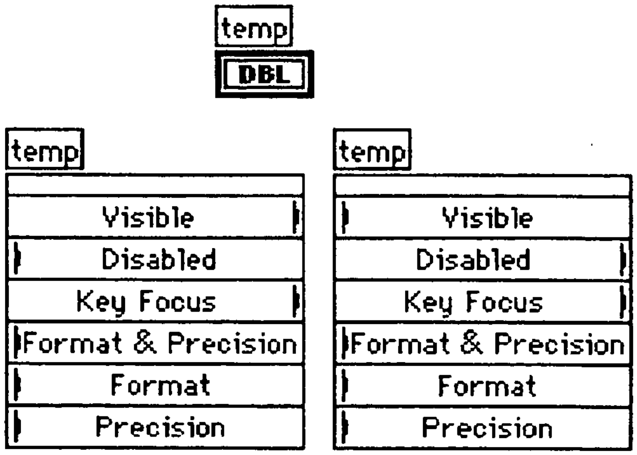


You associate a terminal with a given attribute by clicking with the operating tool on the terminal and selecting an attribute from the Attribute Node pop-up menu.



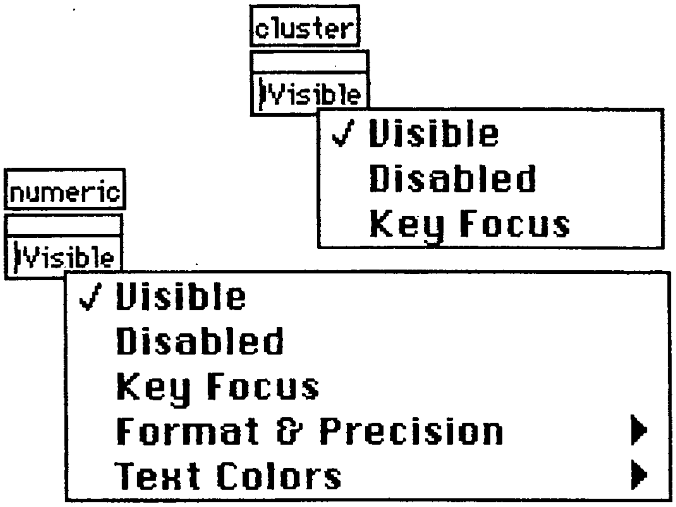


You can create more than one Attribute Node by cloning an existing node, or by selecting the **Create Attribute Node** option again. Each node can have both read and write attribute terminals. You cannot copy and paste an attribute node with the **Edit** menu commands, however.



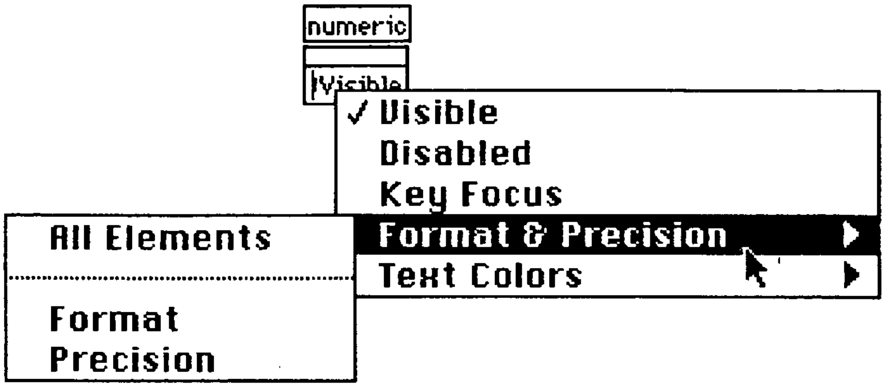
Using Attribute Nodes

You can create Attribute Nodes for any control on a front panel. If the control is an array or cluster, then 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 popup menu. The attributes for a cluster and for a numeric control inside the cluster are shown in the following illustration.

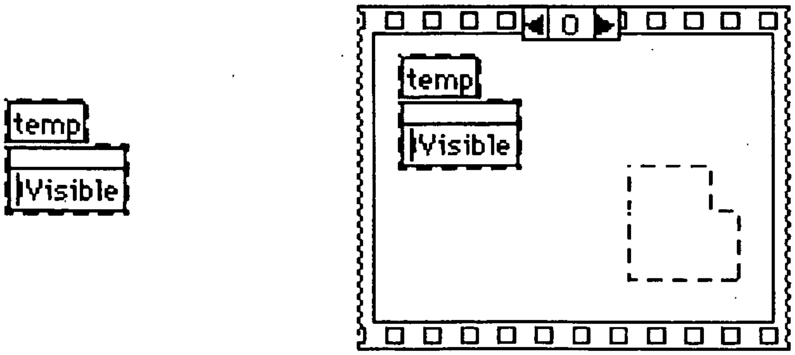


Some controls, such as the graph, have a large number of attributes you can read or set. Some of these attributes are grouped into categories, listed in submenus, such as the **Format & Precision** category for a numeric. You can choose to set all of the attributes at once by selecting the **All Elements** 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.

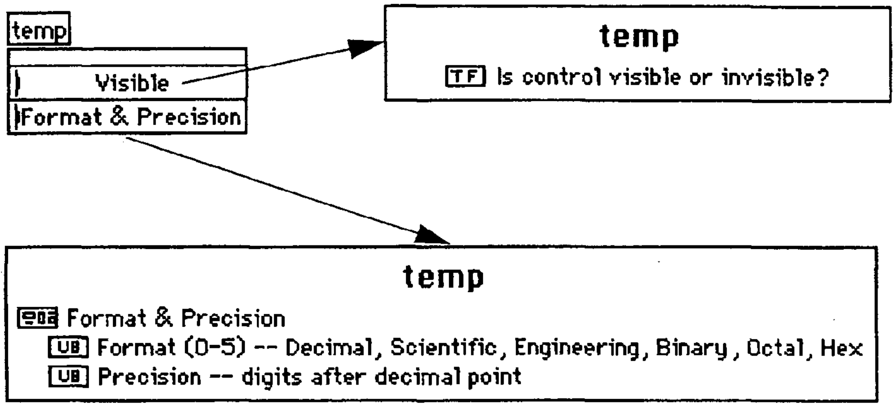


After you create an Attribute Node, the **Find Control** and **Find Terminal** options of the terminal and control popup menus change to submenus that help you find Attribute Nodes as well. In the same way, the Attribute Node has options to find the control and the terminal it is associated with. If you select **Find Attribute Nodes**, all Attribute Nodes on the diagram will be highlighted. If Attribute Nodes are present in different frames of a structure, then one of the nodes will be displayed and highlighted, while a highlighted outline for the hidden Attribute Node indicates its location. This is shown in the following illustration, in which two of three Attribute Nodes are in different frames of a sequence structure. The sequence changes to display two of the nodes, while the other node's location is shown by the highlighted outline.



Attribute Help

If you do not know the meaning of a given attribute, you can use the help window to find out about the attribute meaning, 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.



temp

TF Is control visible or invisible?

temp

Format & Precision
U8 Format (0-5) -- Decimal, Scientific, Engineering, Binary, Octal, Hex
U8 Precision -- digits after decimal point

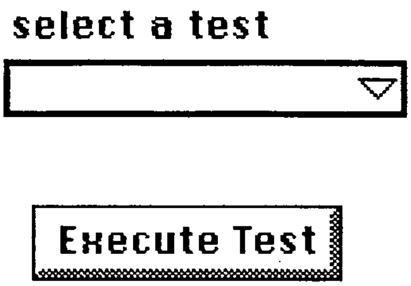
Examples

Included in your distribution software are example programs which illustrate how to set the scales of a graph programmatically, 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.

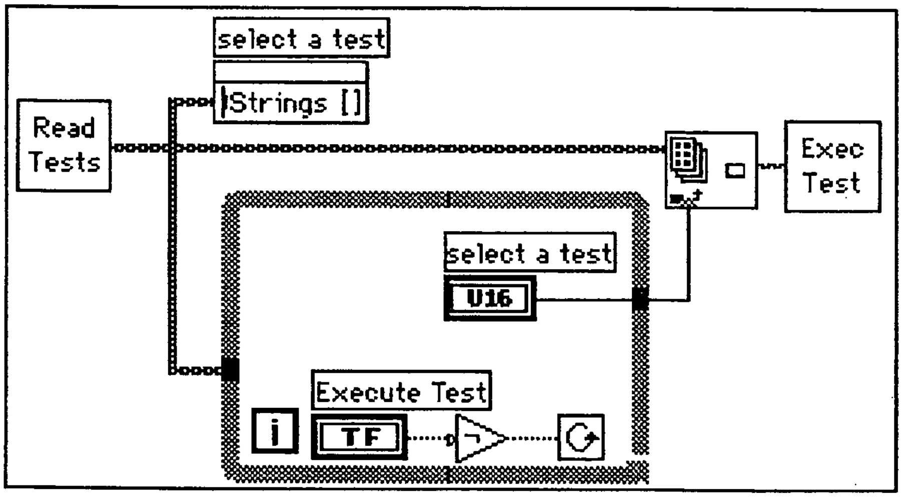
Setting the Strings of a Ring Control

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.

In the following example, the user is presented with a panel that has a ring control which will display a list of tests. The user will select a test and then press the Execute Test button to continue.



The block diagram shown below 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 ring control's numeric value is read and then passed to a VI which executes the test.



Selectively Presenting the User with Options

As a user makes selections, you may want to present him with other options.

One way to do this is to use pop-up subVIs. You can create subVIs which have the options that you want to present the user. By using the **Show Front Panel when Called** and **Close Afterwards if Originally Closed** options of a the **VI Setup** option when you create your subVI, you can have one of these subVIs open when called.

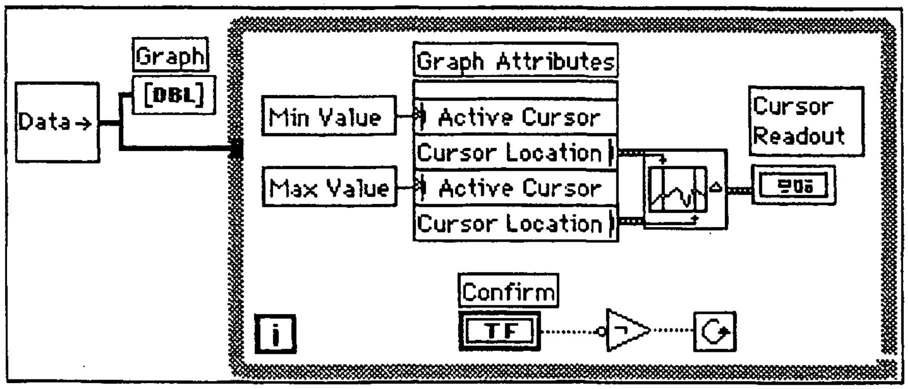
Another method for presenting options is to use the `Visible` option of Attribute Nodes to selectively show and hide controls.

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 multiplot 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.



Available Attributes

Attributes for the various controls are summarized in the following sections.

Base Attributes

A base set of attributes is available for all controls. These attributes include the visibility of a control, whether it is disabled, and 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.

TF **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.

Attributes for Digital Numeric Controls and Color Numeric Controls

In addition to the base attributes, digital numeric controls and color numeric controls have the following attributes:

▣ Format & Precision Cluster of the format and precision attributes.

U8 Format The format for a numeric can be one of the following five values.

- 0 - Decimal
- 1 - Scientific
- 2 - Engineering
- 3 - Binary
- 4 - Octal
- 5 - Hexadecimal

	U8	Precision	Precision is the number of digits displayed after the decimal point.
▣	Text colors	Color numeric value describing the color for the text.	
	U32	Text Color	
	U32	Background Color	

Attributes for Rotary, Slide and Fill Controls

In addition to the base attributes, rotary, slide and fill controls have the following attributes:

I32	Active Slider	Which slide/needle is active.		
▣	Slider Colors	Active slider/needle color.		
	U32	Foreground color for active slider/needle.		
	U32	Background color for active slider/needle.		
▣	Scale Information			
	U32	Scale Style	(0-8) sets scale style as illustrated in control popup menu from top left to bottom right.	
	▣	Format & Precision	Cluster of the format and precision attributes.	
		U8	Format	The format for a numeric can be one of the following five values. • 0 - Decimal • 1 - Scientific • 2 - Engineering • 3 - Binary • 4 - Octal • 5 - Hexadecimal
		U8	Precision	Precision is the number of digits displayed after the decimal point.

	Range	Minimum value, maximum value, and increment.	
	DBL	Minimum	
	DBL	Maximum	
	DBL	Increment	
U8	Mapping Mode	Linear or logarithmic.	
TF	Editable	Determines whether the scale can be edited.	
I32	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	
U32	Fill Color	Fill color for active slider.	

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.

	Format & Precision	Cluster of the format and precision attributes.	
	U8	Format	The format for a numeric can be one of the following five values. • 0 - Decimal • 1 - Scientific • 2 - Engineering • 3 - Binary • 4 - Octal • 5 - Hexadecimal
	U8	Precision	Precision is the number of digits displayed after the decimal point.

▣	Text colors			Color numeric value describing the color for the text.
		U32	Text Color	Color numeric value describing the color of the foreground on which the text is drawn.
		U32	Background color	Color numeric value describing the color of the background on which the text is drawn.
[ ]	Items for Ring and Menu			
		abc	String	

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 array, ramp, and digital display are visible. You can read or set the color array, the out of bounds colors (low and high), and you can select whether the control should interpolate between the specified values. The color array is an array of clusters, where each cluster contains a limit value and the corresponding color.

▣	Text colors			Color numeric value describing the color for the text.
		U32	Foreground Color	Color numeric value describing the color of the foreground for the text.
		U32	Background Color	Color numeric value describing the color of the background for the text.
TF	Array Visible			Is color array visible?
TF	Ramp Visible			Is ramp visible?
TF	DigDisp Visible			Is digital display visible?
▣	Scale Info			Color scale information.
		TF	Interpolate Color	Interpolate between array colors?
		I32	Low Color	For values less than the first array.
		I32	High Color	For values greater than last array.

[ ]	Color Array	Color Array elements.
▦	Value/Color Pair	
DBL	Value	
U32	Color	
U8	Style	Scale Style
▦	Format & Precision	Cluster of the format and precision attributes.
▦	Range	Minimum value, maximum value, and increment.
U8	Mapping Mode	Linear or logarithmic.
TF	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 state 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, then the button returns to FALSE and displays the word run. Pressing on the button while it is TRUE will display the string stop?.

[ ]	Strings[4]	Strings for False, True, True Tracking, and False Tracking states.	
	abc	Boolean Text	
[ ]	Colors [4]	Foreground and background colors for False, True, True Tracking, and False Tracking states.	
	▦	Colors	
		U32	Foreground Color
		U32	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 enable/disable backslash codes, and enable/disable the scrollbar.

U32	Scroll Position	Scroll to line *N* from top of screen.	
▦	Selection	Text selection.	
	U32	Selection Start	Beginning character position.
	U32	Selection End	Ending character position.
TF	\Codes	Are backslash codes enabled?	
TF	Scrollbar	Is vertical scrollbar enabled?	

Attributes for Tables

In addition to the base attributes, Tables have attributes for the control of scroll bar 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 to specific cells, and setting cell foreground and background color.

TF	Index Visible	Show or hide the index digital displays.	
TF	vScroll Vis	Show or hide the vertical scrollbar.	
TF	hScroll Vis	Show or hide the horizontal scrollbar.	
TF	Row Headers Vis	Show or hide the row headers.	
TF	Col Headers Vis	Show or hide the column headers.	
U32	Selection Color	Set the color used to draw the current data selection.	
⊟0a	Edit position	Set the row/column of the current text entry location.	
	I32	Row	
	I32	Column	
⊟0a	Index Values	set the index value of the top left visible cell.	
	U32	Row	
	U32	Column	
⊟0a	Selection Start	Set start of data selection range.	
	I32	Row	Row number.
	I32	Column	Column number.
⊟0a	Selection Size	Number of elements in data selection.	
	I32	Row	Size 0 specificies an insertion point.
	I32	Column	Size 0 specificies an insertion point.
[ ]	Row Headers	Array of strings	
	abc	Header string.	
[ ]	Column Headers	Array of strings	
	abc	Header string.	

▤	Cell Selection	Cell from/to which to read/write cell specific attributes.	
	I32	Row	Row number.
	I32	Column	Column number.
U32	Cell FG Color	Foreground color for cells.	
U32	Cell BG Color	Background color for cells.	

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.

▤	Selection	Text selection.	
	U32	Selection Start	Beginning character position.
	U32	Selection End	Ending character position.

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).

TF	Index Visible		
[ ]	Index Values		
	U32	Index	
[ ]	Selection Start []		
	U32	Index	
[ ]	Selection Size []	Number of elements in array selection.	
	U32	Size	Size 0 specificies an insertion point.

Attributes for Clusters

The cluster has only the base attributes.

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, scroll bar 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 plot's color, interpolation, point style and line style.

TF	Legend Visible	Is the chart legend visible?	
TF	Palette Visible	Is the chart palette visible?	
TF	Scroll Bar Visible	Is the chart scrollbar visible?	
TF	Display(s) Visible	Is the chart digital display visible?	
TF	Transpose Array	Transpose data before plotting?	
⊟0a	Paper Colors	Colors for plotting surface.	
	U32	FG Color	Foreground color for plotting surface.
	U32	BG Color	Background color for plotting surface.
⊟0a	Grid Colors	Grid colors.	
	U32	X Color	Color for X grid.
	U32	Y Color	Color for Y grid.
⊟0a	X Scale Info		
	U8	Scale Fit	Fit x scale to data (0-2) never, once, always.
	U8	Style	X scale style.

		⊟0⊡	Format & Precision			Cluster of the format and precision attributes for the X scale.
		⊟0⊡	Range			X scale range values.
				DBL	Minimum	X scale minimum range value.
				DBL	Maximum	X scale maximum range value.
				DBL	Increment	Increment between X markers.
		U8	Mapping Mode			Linear or logarithmic.
		TF	Editable			Determines whether the scale can be edited.
⊟0⊡	Y Scale Info					
		U8	Scale Fit			Fit Y scale to data (0,1,2) never, once, always.
		U8	Style			Y scale style.
		⊟0⊡	Format & Precision			Cluster of the format and precision attributes for the Y scale.
		⊟0⊡	Range			Y scale range values.
				DBL	Minimum	Y scale minimum range value.
				DBL	Maximum	Y scale maximum range value.
				DBL	Increment	Increment between Y markers.
		U8	Mapping Mode			Linear or logarithmic.
		TF	Editable			Determines whether the scale can be edited.
U32	Active Plot					Which plot is active.
abc	Plot Name					Name of active plot.

▣	Plot Attributes	Attributes of active plot.	
	U32	Plot Color	Color of active plot.
	U8	Plot Interpolation	Interpolation for active plot.
	U8	Point Style	Point style for active plot.
	U8	Line Style	Line style for active plot (0-4) as shown in the pop-up menu from top to bottom.
U32	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.	
	U32	Value.	

Attributes for the Waveform Graph

Nearly identical to the waveform chart, the waveform graph adds Smooth Update, and the cursor attributes.

TF	Smooth Update	Produce smooth (double buffered) updates?	
U32	Active Cursor	Which cursor is active.	
▣	Cursor Attributes	Attributes for active cursor.	
	abc	Cursor Name	Name of active cursor.
	U32	Cursor Color	Color of active cursor.
	U8	Cursor Cross Hair Style	Grid style for active cursor (0-8) as shown in the cursor pop-up menu from top left to bottom right.
	U8	Cursor Point Style	Point style for active cursor (0-16) as shown in the cursor pop-up menu from top left to bottom right.
	TF	Cursor Locked	Is the active cursor locked to a plot?

	U32		Cursor Plot	Plot with which the active cursor is associated.
	U32		Cursor Index	Index of point to which the active cursor is locked.
	▣		Cursor Position	Position of active cursor.
		DBL	Cursor X	Horizontal position of active cursor.
		DBL	Cursor Y	Vertical position of active cursor.
[ ]			Cursor List	List of cursors.

Attributes for the XY Graph

The XY graph attributes are identical to the waveform graph attributes.

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.

TF		Ramp Visible	Is the ramp visible?
TF		Array Visible	Is the color array visible?
▣		Z Scale Info	Color (Z) scale information.
	TF	Interpolate Color	Interpolate between array colors?
	U32	Low Color	Color for values less than the last array element.
	U32	High Color	Color for values greater than the last array element.
	U8	Scale Fit	Fit Z scale to data (0-2) never, once,always.

	[ ]			Color Array	
		▤		Value/Color pair	
			DBL	• Value	
			U32	• Color	
	U8			Style	Z scale style.
	▤			Format & Precision	Cluster of the format and precision attributes for the Z Scale.
	▤			Range	Z scale range values.
	U8			Mapping Mode	Linear or logarithmic.
	TF			Editable	Determines whether the scale can be edited.
TF				Ignore Array	Use colormap instead of array elements.
[ ]				Color Table	Colormap 256 colors.
		U32		Color	
U8				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.
		U32		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.

Chapter 16

Global and Local Variables

This chapter 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. The difference between local and global variables is where the data is stored. In a global variable, data is stored in the front panel items for that global variable. In a local variable, data is stored in items on the front panel of the VI in which that local variable has been placed.

Global and local variables are advanced topics in LabVIEW. Be sure to study this chapter carefully before using these variables.

Global Variables

A global variable is a built-in LabVIEW object. You define a global variable 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.

GLOB
global menu icon

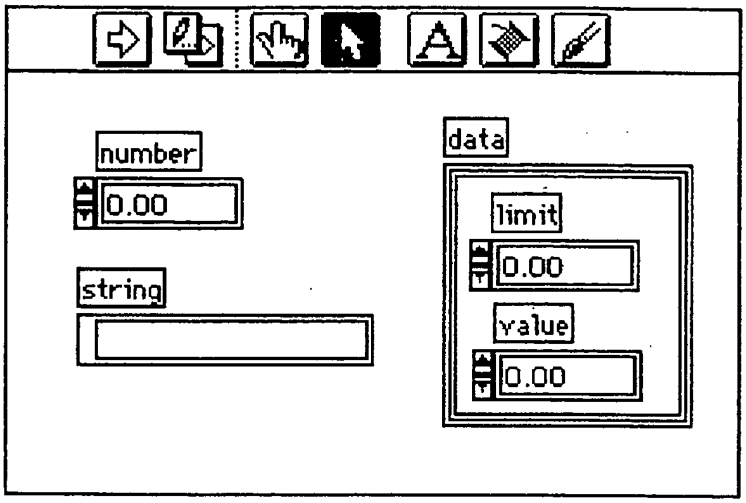
To create a global variable, select the **Global Variable** option from the **Structs & Constants** menu.

A node for the global appears on the diagram.

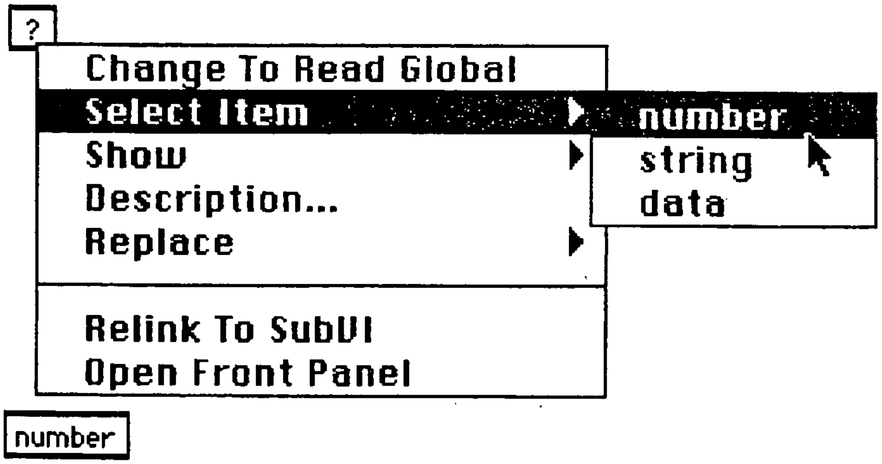
?
global node

Double-click on the node to open a panel window. 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 globals: 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 from the **Select Item** menu, or by clicking on the node with the operating tool and selecting the item you want.



You can either write to a global or read from the global. You select which by using either the **Change to Read Global** or the **Change to Write Global** options of the global variable popup menu.

After you define a global variable, you can place it in other VIs using the **VI...** option of the **Functions** menu. If you select a global variable from the file dialog box, LabVIEW will place the global variable node on the diagram. You can also clone a global, or copy and paste a global.

Local Variables

A local variable is similar to a global variable, except the data storage is one of the controls or indicators on the front panel of your VI instead of on a global variable front panel. Writing to a local variable has the same resulr 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.

LOCAL

local variable menu icon

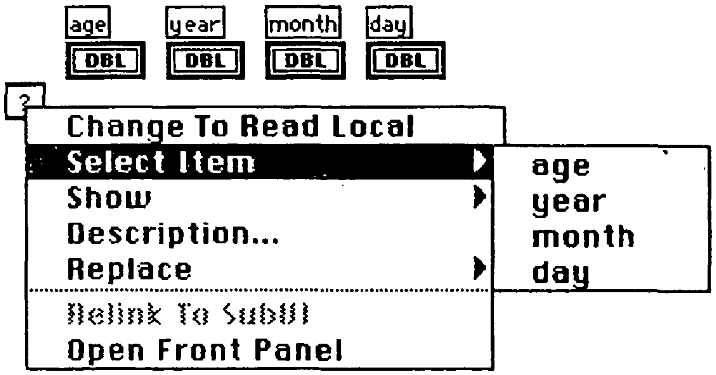
To create a local, select the **Local Variable** option from the **Structs & Constants** menu.

age ?

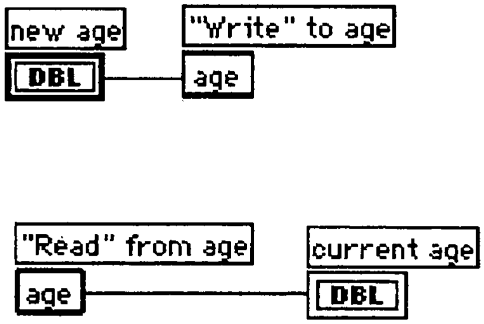
local variable nodes

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 will appear 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.



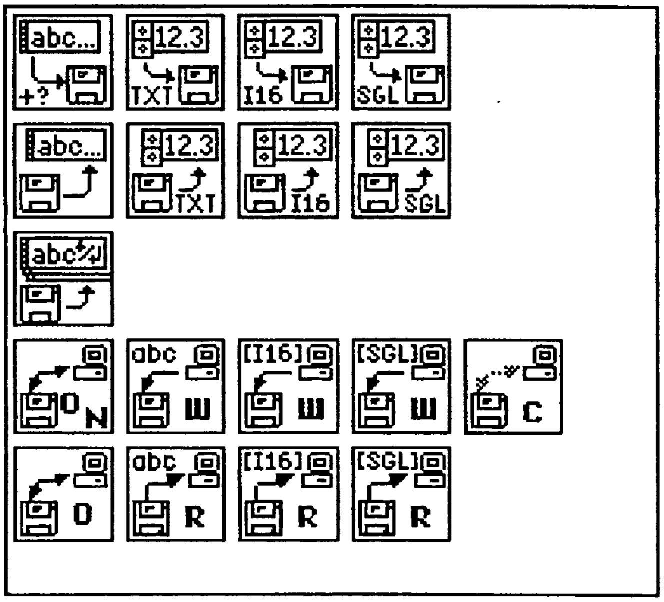
Be careful to sequence the access to local variables or global variables so that you get the results you want, as in the example above. There is no guarantee that the `"Write" to age` will occur before the `"Read" from age` if you do not create the proper sequencing yourself.

The local variable does not track name changes on the front panel. If you change the name of a control or indicator on the front panel, you must change the local variable manually to reflect the new front panel object name.

Chapter 17

File VIs

This chapter describes the utility VIs that perform high- and intermediate-level file I/O operations. The following illustration shows the **File** palette, which you access from the **UTILITY** submenu of the **Functions** menu.



You can use the file VIs to write or read 1D or 2D arrays of SGL numbers to spreadsheet files, or to write or read 1D or 2D arrays of signed word integers (I16), single-precision numbers, or strings to binary byte stream files. You can use these VIs with high- and intermediate-level data acquisition VIs.

The high-level file VIs described here call the intermediate-level VIs 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.

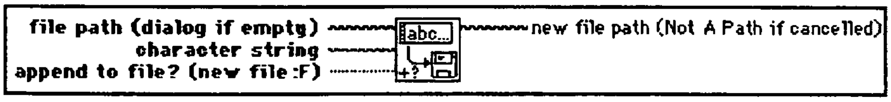
The intermediate-level file VIs call the file functions described in Chapter 8, *File I/O Functions*, of the *LabVIEW Function Reference Manual.* These VIs perform error detection in addition to their other functions.

These VIs use three terminal label styles in the VI help window to distinguish the importance of each input and output. Labels in **bold** font are all you need to use in many applications. You use labels in plain text less often, and labels enclosed in [ ] brackets least of all.

File VI Descriptions

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 beforehand and closes it afterwards.


file path (dialog if empty)
character string
append to file? (new file :F)
abc...
new file path (Not A Path if cancelled)

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 the user cancels the dialog.

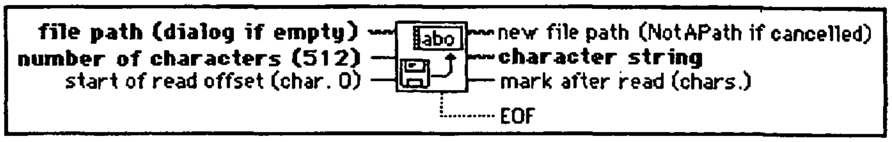
character string is the data the VI writes to the file.

append to file. Set to TRUE if you want to append the data to a existing file; you can also set it 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 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

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 beforehand and closes it afterwards.


file path (dialog if empty)
number of characters (512)
start of read offset (char. 0)
new file path (NotAPath if cancelled)
character string
mark after read (chars.)
EOF

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 the user cancels the dialog.

number of character is the maximum number of characters the VI reads. The VI reads fewer characters if it reaches the EOF first. The default value is 512 characters.

start of read offset is the position in the file, measured in characters (or bytes), at which the VI begins reading.

new file path 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 dialog. new file path returns Not A Path if the user selects Cancel from the dialog box.

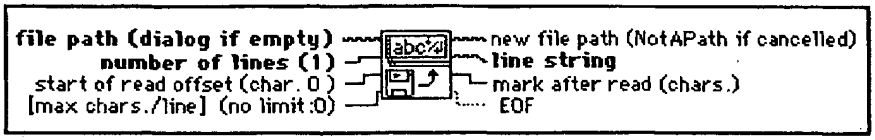
characters string is the data read from the file.

mark after read is the location of the file mark after the read; it points to the character in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of 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 beforehand and closes it afterwards.


file path (dialog if empty)
number of lines (1)
start of read offset (char. 0)
[max chars./line] (no limit:0)
new file path (NotAPath if cancelled)
line string
mark after read (chars.)
EOF

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 the user cancels the dialog.

number of lines is the maximum number of lines the VI reads. The default value is one line. 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 EOF; or a string that has the maximum line length specified by the max characters per line input.

start of read offset 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.

new file path 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

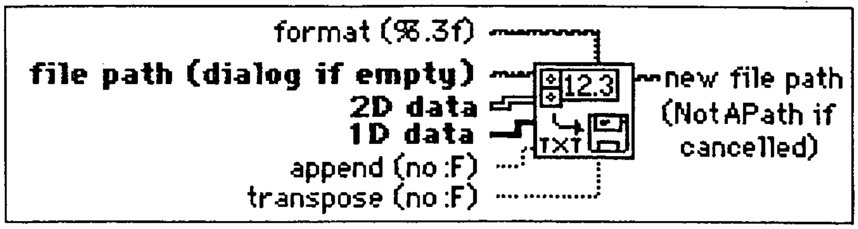
line string is the data read from the file.

mark after read is the location of the file mark after the read; it points to the character in the file following the last character read.

EOF? is TRUE if you attempt to read past the end of the file.

Write To Spreadsheet File

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



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 the user cancels the dialog.

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 transposing it and converting it to a string and writing it to the file. If transpose? is FALSE, each call to this VI creates a new line or row in the file.

append to file? Set to TRUE if you want to append the data to a existing file; you can also set it 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.

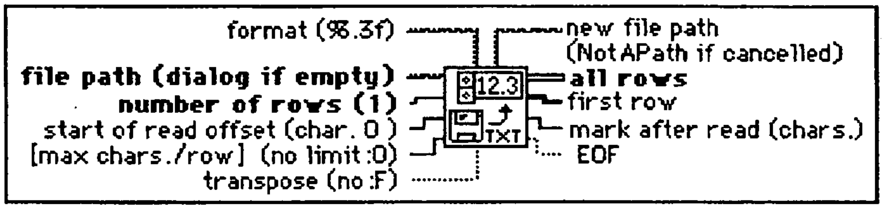
transpose? Set TRUE to transpose the data before converting it to a string. The default value is FALSE.

format 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 in Chapter 6, *String Functions*, of the *LabVIEW Function Reference Manual.*

new file path 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

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 beforehand and closes it afterwards. You can use this VI to read spreadsheet file saved in text format.



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 the user cancels the dialog.

number of rows is the maximum number of rows or lines the VI reads. The default value is one row. 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 EOF; or a string that has the maximum line length specified by the max characters per row input.

start of read offset is the position in the file, measured in characters (or bytes), at which the VI begins reading.

max characters per row 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? Set TRUE to transpose the data after converting it from a string. The default value is FALSE.

abc **format** 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 in Chapter 6, *String Functions*, of the *LabVIEW Function Reference Manual.*

new file path 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 dialog. new file path returns Not A Path if the user selects Cancel from the dialog box.

[SGL] **all rows** is the data read from the file in the form of a 2D array of single-precision numbers.

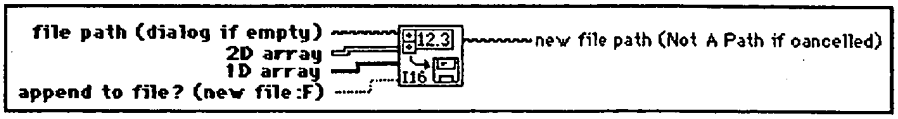
[SGL] **first row** is the first row of the **all rows** array in the form of a 1D array of single-precision numbers. You can use this output when you want to read one row into a 1D array.

I32 **mark after read** is the location of the file mark after the read; it points to the character in the file following the last character read.

TF EOF? is TRUE if you attempt to read past the end of the 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 beforehand and closes it afterwards. You can use this VI to write unscaled or binary data from data acquisition VIs.


file path (dialog if empty)
2D array
1D array
append to file? (new file :F)
12.3
I16
new file path (Not A Path if oancelled)

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 the user cancels the dialog.

[I16] **2D data** contains the signed word integers the VI writes to the file if 1D data is not wired or empty.

[I16]

1D data contains the signed word integers the VI writes to the file if this input is not empty.

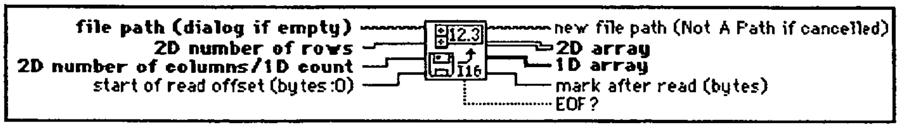
TF

append to file? Set to TRUE if you want to append the data to a existing file; you can also set it 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 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

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 beforehand 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)
2D number of rows
2D number of columns/1D count
start of read offset (bytes:0)
12.3
I16
new file path (Not A Path if cancelled)
2D array
1D array
mark after read (bytes)
EOF?

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 the user cancels the dialog.

I32

2D number of rows is the number of rows to create if the data is to be returned in the 2D array output. If the value is 0 (default), the data is returned in **1D array**.

I32

2D number of columns/1D count is the number of columns to create if the data is to be returned in the 2D array output, provided that **2D number of rows** is greater than 0; otherwise, this is the number of signed word integers to read and return in**1D array**.

I32

start of read offset is the position in the file, measured in bytes, at which the Vl begins reading.

new file path 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

[I16] **2D array** contains the signed word integers read from the file if **2D number of rows** is greater than 0; otherwise, this output is empty.

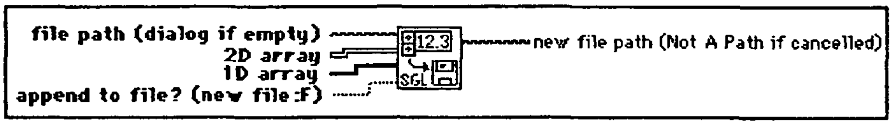
[I16] **1D array** contains the signed word integers read from the file if **2D number of rows** equal 0; otherwise, this output is empty.

I32 **mark after read** is the location of the file mark after the read; it points to the byte in the file following the last byte read.

TF EOF? is TRUE if you attempt to read past the end of file.

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 beforehand and closes it afterwards. You can use this VI to write scaled data from data acquisition VIs.


file path (dialog if empty)
2D array
1D array
append to file? (new file:F)
12.3
SGL
new file path (Not A Path if cancelled)

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 the user cancels the dialog.

[SGL] **2D data** contains the single-precision numbers the VI writes to the file if **1D data** is not wired or empty.

[SGL] **1D data** contains the single-precision numbers the VI writes to the file if this input is not empty.

TF **append to file?** Set to TRUE if you want to append the data to a existing file; you can also set it 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 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

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 beforehand 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 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 the user cancels the dialog.

I32 **2D number of rows** is the number of rows to create if the data is to be returned in the 2D array output. If the value is 0 (default), the data is returned in **1D array.**

I32 **2D number of columns/1D count** is the number of columns to create if the data is to be returned in the 2D array output, provided that **2D number of** rows is greater than 0; otherwise, this is the number of single precision numbers to read and return in**1D array.**

I32 **start of read offset** is the position in the file, measured in bytes, at which the VI begins reading.

new file path 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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

[SGL] **2D array** contains the single-precision numbers read from the file if **2D number of rows** is greater than 0; otherwise, this output is empty.

[SGL] **1D array** contains the single-precision numbers read from the file if **2D number of** rows equal 0; otherwise, this output is empty.

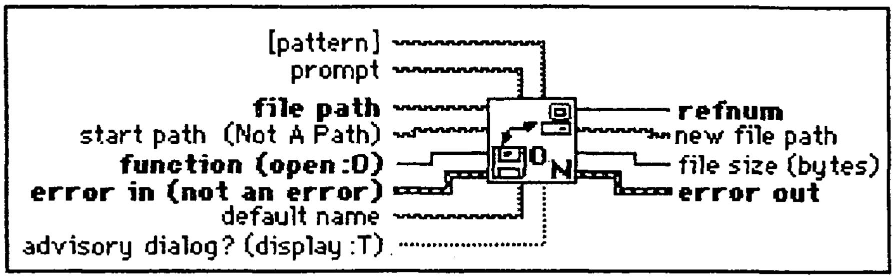
I32 **mark after read** is the location of the file mark after the read; it points to the byte in the file following the last byte read.

TF

EOF? is TRUE if you attempt to read past the end of file.

Open/Create/Replace File

Opens an existing file, creates a new file, or replaces an existing file, programmatically or interactively using dialog. 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+ VIs described below.



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 the user cancels the dialog.

I16

function is the operation to perform:

0: opens an existing file. Error 7 occurs if the file cannot be found.
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. Replacement is done by opening the file and setting EOF to 0. Error 43 occurs if the user elects to disallow the replacement using dialog.
3: creates a new file. Error 8 occurs if the file already exists.

TF

advisory dialog? Set 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.

abc

prompt is the message that appears below the list of files and directories in the file dialog.

abc **default name** is the initial file name that appears in the selection box of the file dialog.

start path is the path name to the initially displayed directory in a file dialog. The default value is Not A Path, which is the path to the last directory shown in a file dialog.

abc **pattern** is the match pattern specification to display only certain types of files or directories. See the description of the File Dialog function for more information.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. The VI executes normally only if no incoming error exists; otherwise it merely passes the error in value to error out.

> TF **status** is TRUE if an error occurred; the default value is FALSE.
>
> I32 **code** is the error code; the default value is 0.
>
> abc **source** is in most cases the name of the VI or function that produced the error; the default value is an empty string.

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 dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

refnum is the reference number of the open file. The value is Not A Refnum if the file cannot be opened.

I32 **file size** is the size of the file in bytes; it is also the location of the end of file.

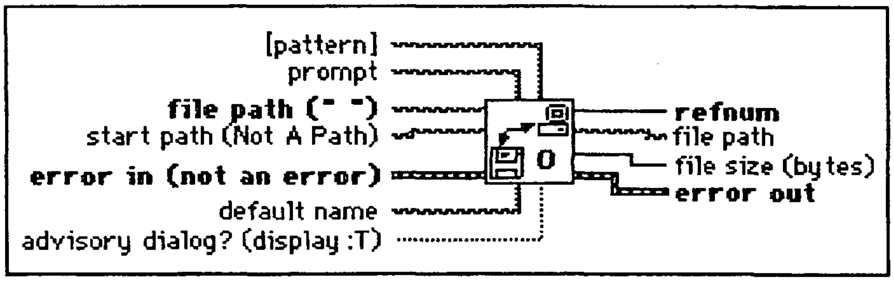
error out is a cluster of three elements (**status, code,** and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

> TF **status** is TRUE if an error occurred.
>
> I32 **code** is the error code.

source is in most cases the name of the VI or function that produced the error.

Open File+

Opens an existing file programmatically or interactively using dialog. You can optionally specify a dialog prompt, default file name, start path, or filter pattern. Error 7 occurs if the file does not exist. Use this VI with the intermediate level Read File+ VIs described below.



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 the user cancels the dialog.

advisory dialog? Set TRUE (default) if you want a dialog if the file does not exist.

prompt is the message that appears below the list of files and directories in the file dialog.

default name is the initial file name that appears in the selection box of the file dialog.

start path is the path name to the initially displayed directory in a file dialog. The default value is Not A Path, which is the path to the last directory shown in a file dialog.

pattern is the match pattern specification to display only certain types of files or directories. See the description of the File Dialog function for more information.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. The VI executes normally only if no incoming error exists; otherwise it merely passes the error in value to error out.

TF **status** is TRUE if an error occurred; the default value is FALSE.

I32 **code** is the error code; the default value is 0.

abc **source** is in most cases the name of the VI or function that produced the error; the default value is an empty string.

new file path is the path of the file opened. You can use this output to determine the path of a file that you open using dialog. **new file path** returns Not A Path if the user selects **Cancel** from the dialog box.

refnum is the reference number of the open file. The value is Not A Refnum if the file cannot be opened.

I32 **file size** is the size of the file in bytes; it is also the location of the end of file.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

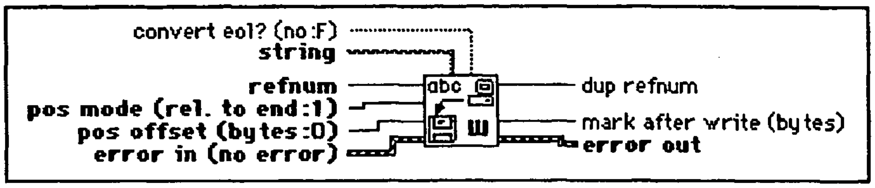
TF **status** is TRUE if an error occurred.

I32 **code** is the error code.

abc **source** is in most cases the name of the VI or function that produced the error.

Write File+ (string)

Writes a string of characters to a byte stream file at the specified location; the default location is at the end of file. The VI does incoming and internal error checking and does not write if an incoming error occurs.



refnum is the reference number of the open file.

pos mode specifies where the write begins relative to pos offset:

0: Beginning of the file plus pos offset.
1: End of the file plus pos offset. This is the default value.
2: Current file mark plus pos offset.

pos offset specifies in characters how far from the location specified by pos mode that the operation begins. The default value is 0.

string is the data the VI writes to the file.

convert eol? converts the LabVIEW end of line marker to the platform specific marker when set to TRUE. When set to false (the default value), no conversion occurs.

error in is a cluster of three elements (**status, code,** and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. 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; the default value is FALSE.

code is the error code; the default value is 0.

source is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

mark after write is the location of the character following the last character written.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

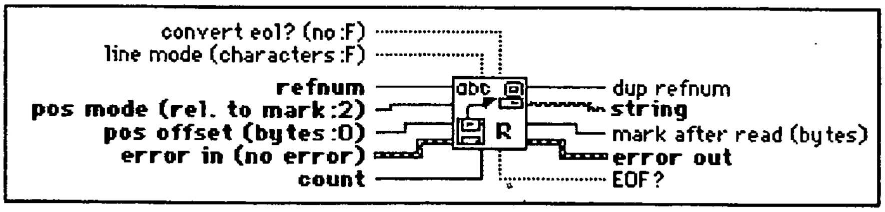
status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

Read File+ (string)

Reads a string of characters from a byte stream file at the specified location; the default location is at the current mark. The VI does incoming and internal error checking and does not read if an incoming error occurs.



refnum is the reference number of the open file.

pos mode specifies where the reads begins relative to pos offset:

0: Beginning of the file plus pos offset.
1: End of the file plus pos offset.
2: Current file mark plus pos offset. This is the default value.

pos offset specifies in characters how far from the location specified by pos mode that the operation begins. The default value is 0.

count is the number of bytes to read.

line mode Set TRUE to read until a line marker or end of line is encounter or count characters are read. Set FALSE (default value) to read until count characters are read or end of line is encountered. A line mark is a carriage return, a line feed, or a carriage return followed by a line feed.

convert eol? converts the LabVIEW end of line marker to the platform specific marker when set to TRUE. When set to false (the default value), no conversion occurs.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. 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; the default value is FALSE.

code is the error code; the default value is 0.

source is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

character string is the data read from the file.

mark after read is the location of the file mark after the read; it points to the character in the file following the last character read.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

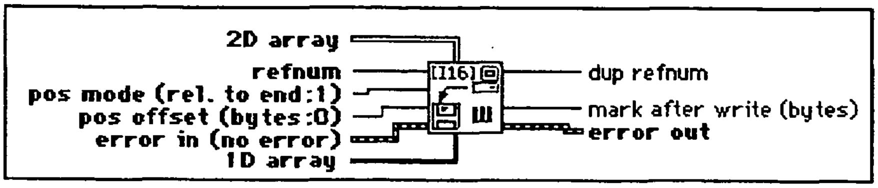
status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

Write File+ [I16]

Writes a 2D or 1D array of signed word integers (I16) to a byte stream file at the specified location; the default location is at the end of file. The VI does incoming and internal error checking and does not write if an incoming error occurs.



refnum is the reference number of the open file.

I16 **pos mode** specifies where the write begins relative to pos offset:

0: Beginning of the file plus pos offset.
1: End of the file plus pos offset. This is the default value.
2: Current file mark plus pos offset.

I32 **pos offset** specifies in bytes how far from the location specified by pos mode that the operation begins. The default value is 0. Byte units rather than integer (2 byte) units for flexibility, for example, so that a file can contain a text header written with Write File+ (string) followed by arrays of number written with Write File+ [I16].

[I16] **2D array** contains the signed word integers the VI writes to the file if 1D array is not wired or is empty.

[I16] **1D array** contains signed word integers the VI writes to the file if this input is not empty.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. The VI executes normally only if no incoming error exists; otherwise it merely passes the error in value to error out.

TF **status** is TRUE if an error occurred; the default value is FALSE.

I32 **code** is the error code; the default value is 0.

source is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

mark after write is the location of the character following the last character written.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

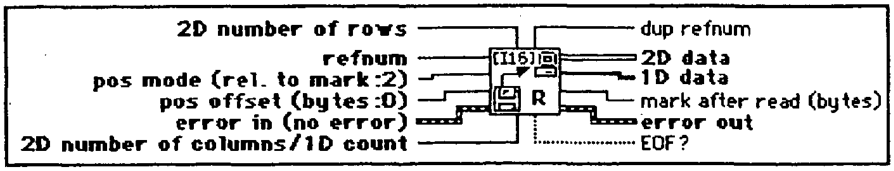
status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

Read File+ [I16]

Reads a 2D or 1D array of signed word integers (I16) from a byte stream file at the specified location; the default location is at the current mark. The VI does incoming and internal error checking and does not read if an incoming error occurs.



refnum is the reference number of the open file.

pos mode specifies where the write begins relative to pos offset:

0: Beginning of the file plus pos offset.
1: End of the file plus pos offset.
2: Current file mark plus pos offset. This is the default value.

I32 **pos offset** specifies in bytes how far from the location specified by pos mode that the operation begins. The default value is 0. Byte units rather than integer (2 byte) units for flexibility, for example, so that a file can contain a text header written with Write File+ (string) followed by arrays of number written with Write File+ [I16].

I32 **2D number of rows** is the number of rows to create if the data is to be returned in the 2D array output. If the value is 0 (default), the data is returned in the 1D array.

I32 **2D number of columns/1D count** is the number of columns to create if the data is to be returned in the 2D array output, provided that **2D number of rows** is >0; otherwise, this is the number of signed word integers to read and return in the 1D array output.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. The VI executes normally only if no incoming error exists; otherwise it merely passes the error in value to error out.

> TF **status** is TRUE if an error occurred; the default value is FALSE.
>
> I32 **code** is the error code; the default value is 0.
>
> abc **source** is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

[I16] **2D array** contains the signed word integers read from the file if **2D number of rows** is greater than 0; otherwise, this output is empty.

[I16] **1D array** contains the signed word integers read from the file if **2D number of rows** equal 0; otherwise, this output is empty.

I32 **mark after read** is the location of the file mark after the read; it points to the byte in the file following the last byte read.

TF **EOF?** is TRUE if you attempt to read past the end of file.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

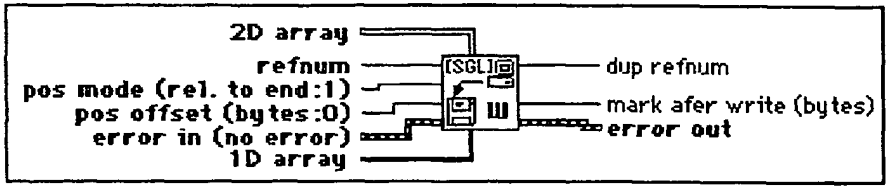
TF **status** is TRUE if an error occurred.

I32 **code** is the error code.

abc **source** is in most cases the name of the VI or function that produced the error.

Write File+ [SGL]

Writes a 2D or 1D array of single-precision numbers (I16) to a byte stream file at the specified location; the default location is at the end of file. The VI does incoming and internal error checking and does not write if an incoming error occurs.



refnum is the reference number of the open file.

I16 **pos mode** specifies where the write begins relative to pos offset:

0: at the beginning of the file plus pos offset.
1: at the end of the file plus pos offset. This is the default value.
2: at the current file mark plus pos offset.

I32 **pos offset** specifies in bytes how far from the location specified by pos mode that the operation begins. The default value is 0. Byte units rather than I16 (2 byte) units for flexibility, for example, so that a file can contain a text header written with Write File+ (string) followed by arrays of number written with Write File+ [I16].

[SGL] **2D array** contains the single-precision numbers the VI writes to the file if 1D array is not wired or is empty.

[SGL] **1D array** contains single-precision numbers the VI writes to the file if this input is not empty.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. 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; the default value is FALSE.

code is the error code; the default value is 0.

source is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

mark after write is the location of the character following the last character written.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

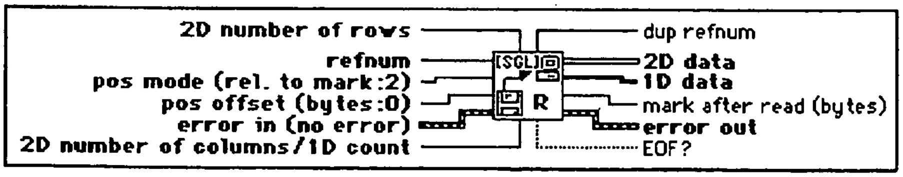
status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

Read File+ [SGL]

Reads a 2D or 1D array of single-precision numbers (SGL) from a byte stream file at the specified location; the default location is at the current mark. The VI does incoming and internal error checking and does not read if an incoming error occurs.



refnum is the reference number of the open file.

I16 **pos mode** specifies where the write begins relative to pos offset:

0: Beginning of the file plus pos offset.
1: End of the file plus pos offset. This is the default value.
2: Current file mark plus pos offset.

I32 **pos offset** specifies in bytes how far from the location specified by pos mode that the operation begins. The default value is 0. Byte units rather than SGL (4 byte) units for flexibility, for example, so that a file can contain a text header written with Write File+ (string) followed by arrays of number written with Write File+ [SGL].

I32 **2D number of rows** is the number of rows to create if the data is to be returned in the 2D array output. If the value is 0 (default), the data is returned in**1D array**.

I32 **2D number of columns/1D count** is the number of columns to create if the data is to be returned in the 2D array output, provided that **2D number of rows** is >0; otherwise, this is the number of signed word integers to read and return in**1D array**.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. The VI executes normally only if no incoming error exists; otherwise it merely passes the error in value to error out.

TF **status** is TRUE if an error occurred; the default value is FALSE.

I32 **code** is the error code; the default value is 0.

abc **source** is in most cases the name of the VI or function that produced the error; the default value is an empty string.

dup refnum is a flow-though parameter with the same value as refnum.

[SGL] **2D array** contains the single-precision numbers read from the file if **2D number of rows** is greater than 0; otherwise, this output is empty.

[SGL] **1D array** contains the single-precision numbers read from the file if **2D number of rows** equal 0; otherwise, this output is empty.

I32 **mark after read** 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.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

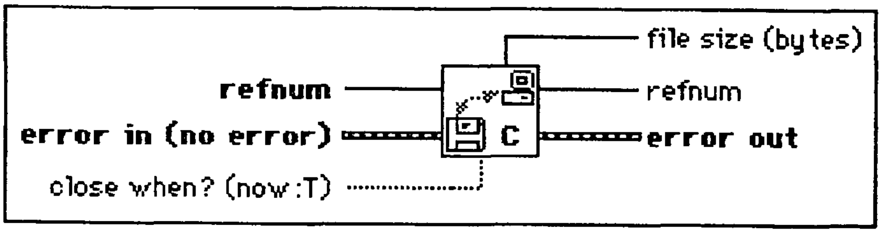
status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

Close File+

Closes the file on command or when in incoming error occurs and returns the file size at closure.



refnum is the reference number of the open file.

close when? Set TRUE (default) when you want to close the file when this VI is called. Set to FALSE when you want to close the file only if this VI encounters an incoming error. This input is useful when you use this VI inside a loop and want to close the file only on the last loop iteration or on an upstream error. In that case, set **close when?** TRUE on the last iteration, and terminate the loop on the last iteration or when this VI returns an error.

error in is a cluster of three elements (**status**, **code**, and **source**) that describe the *upstream* error state, that is, the existence of any error proceeding the execution of the VI. 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; the default value is FALSE.

code is the error code; the default value is 0.

source is in most cases the name of the VI or function that produced the error; the default value is an empty string.

file size is the size of the file in bytes at the time the file is closed; the value is zero otherwise.

dup refnum is a flow-though parameter with the same value as refnum.

error out is a cluster of three elements (**status**, **code**, and **source**) that describe the error state following the execution of the VI. If an incoming error does not exist, the VI executes normally and **error out** describes the VI's error state; if an incoming error exists, **error out** equals error in.

status is TRUE if an error occurred.

code is the error code.

source is in most cases the name of the VI or function that produced the error.

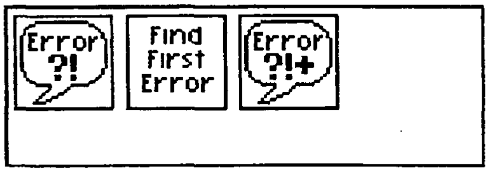
Chapter 18

Error Handler VIs

This chapter describes a set of VIs for managing and reporting errors.

You can connect these VIs to error status terminals of other VIs to test whether an error has occurred. If an error has occurred, these VIs return a text-based description of the error. In addition, you can have these VIs display a dialog box with a description of the error message. You can select either a dialog box with an OK button, or a dialog box with buttons that allow the user to stop or continue execution.

The following illustration shows the **Error Handlers** palette, which you access from the **UTILITY** submenu of the **Functions** menu.



Error Handling

When designing applications, you should consider the possibility of unexpected situations and decide how you are going to handle them. For instance, when writing to a file you should consider the possibility that the operation might fail due to a lack of disk space. If you are designing an application that needs reliable operation, you should check for errors and decide what you want to do when an error occurs.

LabVIEW I/O operations check for errors and return error codes. Error handling is different for every application, so LabVIEW doesn't perform automatic handling of errors. For instance, you may want to shut the application down after receiving an error, or you may just want to notify the user so that he or she can correct the problem.

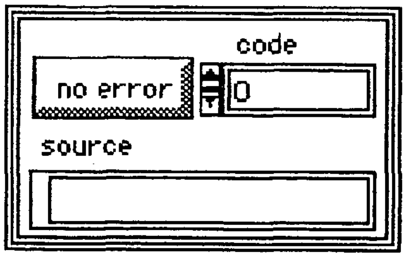
Most of the low-level I/O operations in LabVIEW return an error code, where a non-zero value indicates that an error occurred. Some of the higher level I/O operations incorporate a scheme that makes handling errors easier. Under this scheme, I/O VIs have both an error input and output. If the input indicates an error, then the I/O VI either does nothing or shuts down any I/O operation that it is managing. One advantage of this error input/output design is that you can connect several I/O operations together so that, if an error occurs, LabVIEW will not perform subsequent unnecessary VIs, and the user can handle the error.

Another advantage of this error input/output design is that it allows you to establish the order of a set of I/O operations, even though there may not be any data dependency between the operations. You could establish execution order with a Sequence structure, but the error input/output design allows you to establish the order with all of the operations on the same level of the block diagram.

You can use the error handlers described in this chapter at the end of a sequence of I/O operations to convert the error code information into a description of the error, which can be displayed to the user.

Error Input/Output Cluster

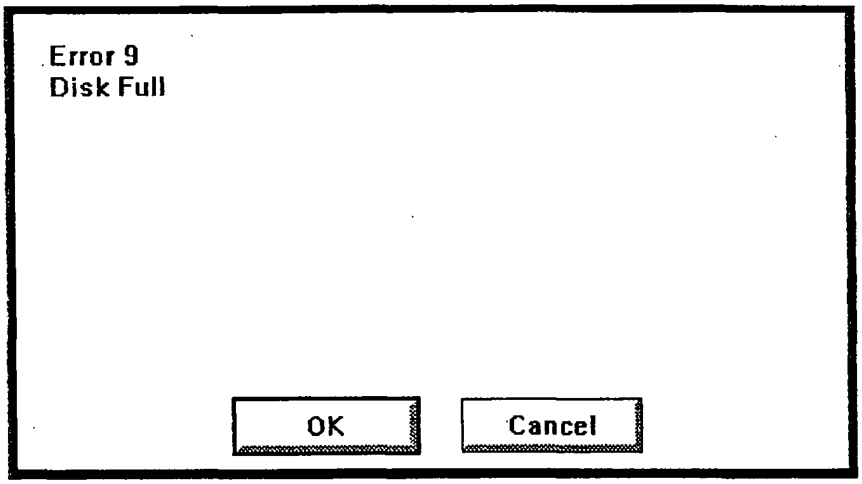
The error input/output used by many of the high-level I/O operations is a cluster containing a Boolean indicating whether an error has been detected, the error code, and a descriptive string that usually contains the name of the source of the error. This cluster is shown in the following illustration.



Error Handler VI Overview

There are three error handlers in the **Error Handlers** palette. These error handlers contain error information about all I/O operations in LabVIEW, including file I/O, data acquisition, GPIB, serial I/O, DDE, and TCP/IP.

The first VI, Simple Error Handler, will take care of the error handling needed by most applications. You can pass the error information to this VI either as the error cluster or as an error code with an optional descriptive string. If **error cluster in** indicates an error, the VI reports that error as output. If not, or if **error cluster in** is not wired, the VI checks **error code** to see if it is non-zero. If it is, the VI converts the error code to a description of the error, if possible. You can also have the VI display a dialog box with the error description and an OK button, or a dialog box with the error description and buttons that allow the user to stop or continue execution, as shown in the following illustration:



Error 9
Disk Full

You use the second VI, Find First Error, when you need to test a set of low-level functions or subVIs that output only a numeric error code rather than the error output cluster. As the name implies, this VI determines which function or subVI first detected an error. It then forms an error output cluster so that the Simple or General Error Handler can report the error to the user.

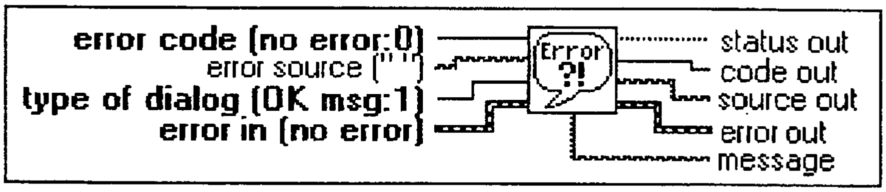
The third VI, General Error Handler, does the same thing as the Simple Error Handler VI, but you can use your own list of user-defined error codes and error descriptions. In addition, with this VI you can treat an error differently from the way it is normally treated. For instance, many I/O operations return an error code if they time out. You may not want to treat a timeout as an error in some cases. You can treat the timeout as an exception to normal error handling so that the error handler will ignore it.

If you have an I/O operation for which the error handlers do not have predefined descriptions, you can pass the error code and description as a user-defined error code to the General Error Handler if you want to display a descriptive error message.

Error Handler VIs

Simple Error Handler

Ascertains whether an error has occurred. If it finds an error, this VI creates a description of the error and optionally displays a dialog box.



I32

type of dialog 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 how the error is handled.

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 allowing the user to either continue or stop. If the user cancels, the VI calls the Stop function to halt execution.

I32

error code is a numeric error code. The VI ignores this value if **error in** indicates an error. Otherwise this value is tested. A non-zero value signifies an error.

abc

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 in describes an error that you want to check. If you leave **error in** unwired, this VI checks **error code** for errors.

TF **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.**

I32 **code** is the error code associated with an error.

abc **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.

TF **status out** is TRUE if the VI found an error.

I32 **code out** is the error code indicated by **error in** or **error code.**

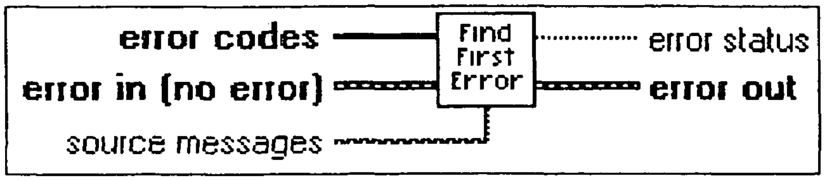
abc **source out** indicates the source of the error.

error out is a cluster containing the same information as in **status out, code out,** and **source out.** It has the same structure as **error in.**

abc **message** describes the error that occurred and the source of the error.

Find First Error

Tests the error status of one or more low-level functions or subVIs that output a numeric error code. If it finds an error, this VI creates an **error out** cluster you can wire to the Simple or General Error Handler to identify the error and describe it to the user.

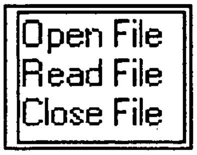


[I32] **error codes** is an array of the numeric error codes assembled from local subVIs or functions. If there is no error indicated in the **error in** cluster, the VI tests these codes in ascending order for non-zero values. If the VI finds a non-zero value, **error out** reflects the error status of that input.

error in is a cluster of elements that describes any previous error you want to check. If it is unwired or indicates no error, the VI checks the **error codes** array instead. See Simple Error Handler for additional information.

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 in a string constant or control.

If you want to test for an error from three local functions (for example, Open File, Read File, and Close File), wire their error outputs to a Build Array function, as elements 0-2 respectively. Use the following string for **source messages:**



Open File
Read File
Close File

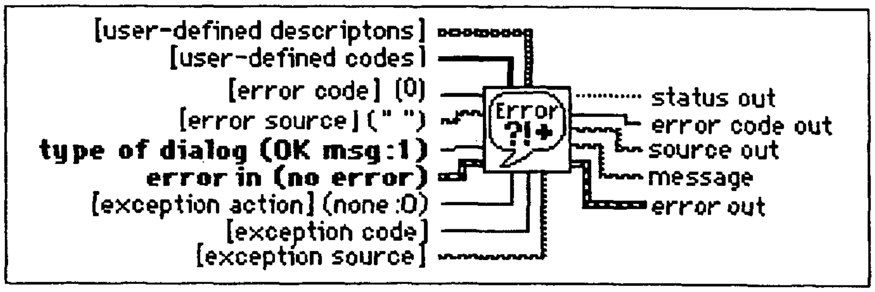
If an error occurs at Read File, the VI picks the second line for the source message of the error out cluster. Use of this input is optional.

error status is TRUE if the **error in** cluster or any of the **error codes** reflects an error.

error out is a cluster that reflects the status of the first error input that tests TRUE. It has the same elements as **error in.**

General Error Handler

Ascertains whether an error has occurred. If an error has occurred, this VI creates a description of the error and optionally displays a dialog box.



I32 **type of dialog** determines what type dialog box will be displayed, if any. Regardless of its value, the VI outputs error information and **message** describing the error.

0: displays no dialog box. This is useful if you want to have programmatic control over how the error is handled.
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 cancels, the VI calls the Stop function to halt execution.

error in describes an error that you want to check. If unwired, this VI checks **error code** for errors.

TF **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**.

I32 **code** is the error code associated with an error.

abc **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.

[abc] **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**.

[I32] **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.

I32 **error code** 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.

abc **error source** is an optional string you can use to describe the source of **error code**. The VI uses the string in **message** if there is an error.

I32 **exception action** 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 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 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**.

I32 **exception code** is the error code that you want to treat as an exception. By default, it is 0.

abc **exception source** is the error message that you want to use to test for an exception. By default, it is an empty string.

TF **status out** is TRUE if an error was indicated.

I32 **code out** is the error code indicated by the **error in** or **error code**.

abc **source out** indicates the source of the error.

abc **message** describes the error code that occurred, the source of the error, and a description of the error.

error out contains the same information as **status out, code out**, and **source out**. It has the same structure as **error in.**

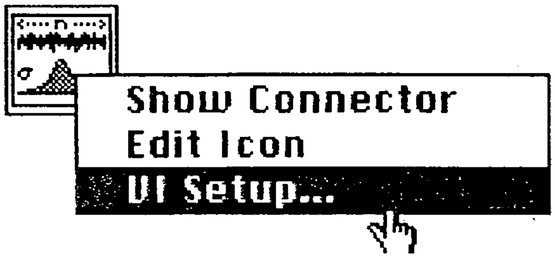
Chapter 19

VI Setup Options

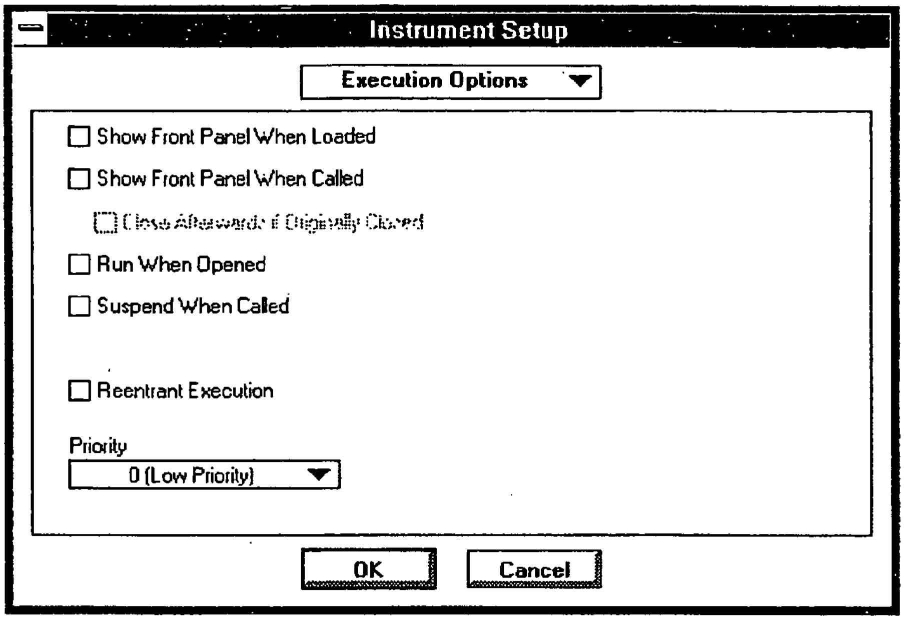
This chapter lists the VI setup options you can use to create special effects such as pop-up panels, simulated controls and indicators, asynchronous execution, reentrant execution, and so on, as well as other unique LabVIEW features.

VI Setup... and SubVI Node Setup...

The VI Setup dialog boxes appear when you select **VI Setup...** from the icon pane pop-up menu on the Panel window of a VI.



To switch between the Execution Options and the Window Options, click on the downward pointing arrowhead in the title box and select the box that displays the options you want to change.



□ Show Front Panel When Loaded

□ Show Front Panel When Called

□ Close Afterwards if Originally Closed

□ Run When Opened

□ Suspend When Called

□ Reentrant Execution

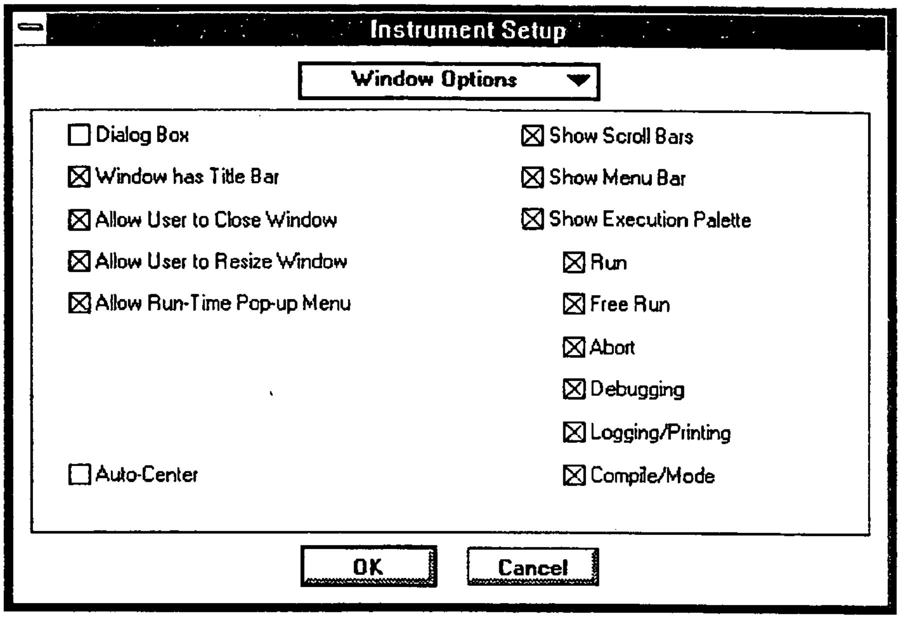
Priority

0 (Low Priority) ▼

The Execution Options apply to the VI when you use it as a subVI. You can use these options to create a pop-up panel for your application, simulate a dialog box, set a break point, enable reentrant execution for the VI, and set the execution priority for the VI. These options apply to every instance of the VI when used as a subVI.

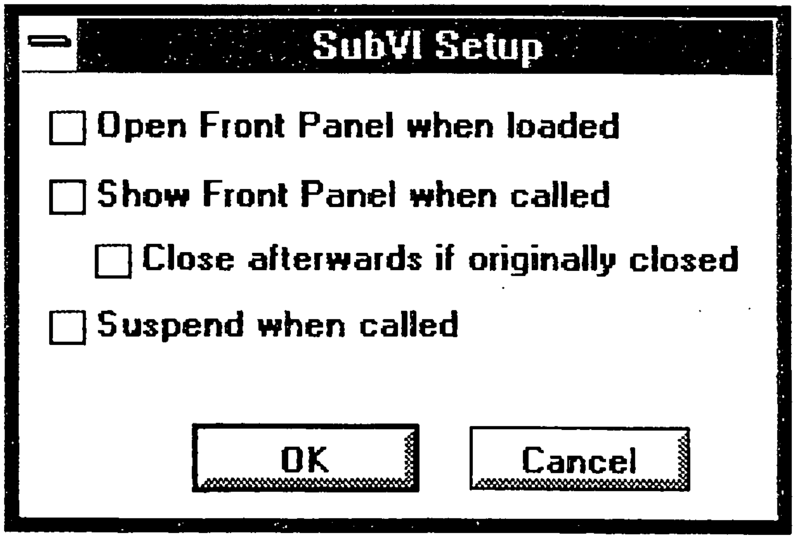
You can also use the **Run When Opened** option for top level VIs.

The Window Options apply to the VI when it is in the run mode, but not 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 forcing the user to respond only to the options presented by the panel.



- □ Dialog Box
- ☒ Window has Title Bar
- ☒ Allow User to Close Window
- ☒ Allow User to Resize Window
- ☒ Allow Run-Time Pop-up Menu
- □ Auto-Center
- ☒ Show Scroll Bars
- ☒ Show Menu Bar
- ☒ Show Execution Palette
  - ☒ Run
  - ☒ Free Run
  - ☒ Abort
  - ☒ Debugging
  - ☒ Logging/Printing
  - ☒ Compile/Mode

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. All options in this box apply *only to the node from which you select this option.*



- □ Open Front Panel when loaded
- □ Show Front Panel when called
  - □ Close afterwards if originally closed
- □ Suspend when called

The following sections discuss using the VI setup options.

Disabling Front Panel Features

You can control user access to LabVIEW features using the Window Options of the **VI Setup...** dialog box. These options prevent users from accidentally interfering with the execution of a VI, and also force the user to respond to a panel appropriately. By using these options, you can also make your pop-up panels look like dialog boxes, controls, or indicators.

For example, deselecting the **Allow User to Close Window** check box in the **VI Setup...** Window Options dialog box hides the close box for the VI front panel, and dims the **Close** option in the **File** menu. With this VI setup option, you can ensure that an operator cannot inadvertently close a run-time or pop-up VI front panel.

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.

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.

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.

The rest of the options determine which execution palette buttons appear on the front panel. You can hide individual buttons from the execution palette, or the entire palette with the **Show Execution Palette** option.

⚠ **Caution:** ***If you hide the menu bar and the execution palette, there is no visible means of changing from run mode back into edit mode. You can use the <Ctrl-M> hot key that corresponds to the Change to Edit Mode item from the Operate menu to change the VI back to edit mode, where the menu bar and palette will be visible.***

All these options apply to the execution palette while it is in run mode, but not while it is in edit mode.

Asynchronous Execution and VI Priority

An *asynchronous* node can execute simultaneously with other nodes. A synchronous node must run to completion before any other node can begin.

A CPU can execute code from only one node at a time. So in this context, simultaneously means executing portions of multiple code segments during a certain time interval rather than at the same instant in time. LabVIEW contains a multitasking execution system that interleaves the execution of all nodes that have received all of their input data. The LabVIEW *priority scheduler* automatically gives each node a small amount of time in turn, according to their priorities. Nodes execute for a time, then rotate to the end of the queue of nodes, and the next node in line executes. When a node completes execution, it is removed from the queue.

Structures, I/O functions, timing functions, and subVIs that you build execute asynchronously. Code Interface Nodes (CINs) and all computational functions execute synchronously. Most analysis VIs contain CINs and therefore execute synchronously. For example, an FFT executes to completion without interleaving execution of any other node, regardless of how long the FFT takes.

Normally, you need not be concerned about the asynchronous nature of your VIs. You can think of portions of a diagram as executing in parallel, and multiple VIs as running in parallel. With the LabVIEW multitasking ability, you can edit a VI while others run, or you can single step through a VI while others execute at full speed. Also, if you inadvertently build a VI with an infinite loop, it will not lock up the computer or prevent other VIs from executing.

In some cases, you may want to give a node priority over other nodes. You change a node's priority through the Execution Options of 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 scheduler queue contains two VIs of each priority level, the priority 3 VIs share execution time exclusively until both of them finish. Then, the level 2 VIs take turns executing until they both finish, and so on. In this way you can create a block diagram that gives execution preference to certain VIs.

The highest priority available is called subroutine priority, and it is special in several ways. When a VI is running at subroutine priority, no other VI can run until the subroutine priority VI has finished. A subroutine VI execution is streamlined, so much so that the front panel is not used at all. Watching a subroutine VI front panel will reveal nothing about its execution.

A subroutine VI may call other subroutine VIs, but may not call a VI with any other priority. Use subroutine VIs in situations where you want to put a simple computation with no interactive elements into a reusable VI with reduced call overhead.

Reentrant Execution

Under normal circumstances, LabVIEW cannot interleave execution of multiple calls to the same subVI. A normal subVI is *serially reusable.* This is important, for example, when the subVI communicates with an I/O device. Otherwise, the communications could get scrambled. A subVI that can execute multiple calls in parallel is *reentrant.* Each call to a reentrant subVI maintains its own separate copy of data. Reentrant execution is useful in the following three situations.

- When an instrument driver VI controls and communicates with a physical device and must control *multiple* identical instruments.
- 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 opening a VI or subVI when loaded, opening a subVI when called, execution highlighting, and 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.

Controlling Multiple Identical Instruments with a Driver VI

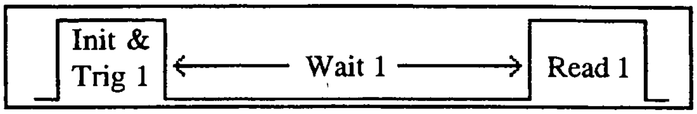
Suppose you have a voltmeter that you control using a VI named DVM. To take a measurement, DVM uses the following sequence.

1. Write to the instrument to configure and trigger it.

2. Wait for the instrument to signal that it has completed the measurement by requesting service.

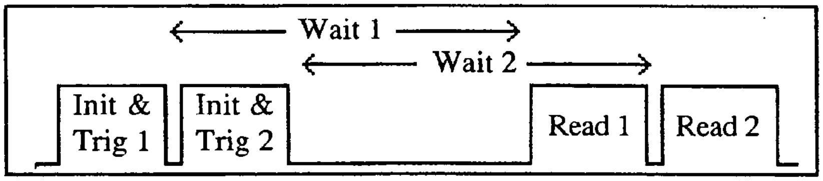
3. Read the measured value.

The following drawing represents the time required to complete the voltmeter operations.



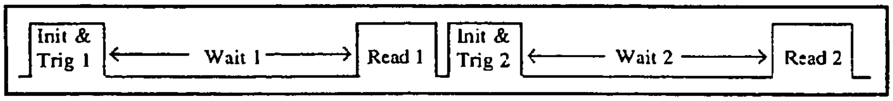
Between Steps 1 and 3, LabVIEW waits while the instrument makes the measurement. The wait operation is asynchronous, so LabVIEW can perform other asynchronous operations in parallel.

Now suppose you have two identical voltmeters which you are using to measure two signals at the same time. You could make a copy of DVM and save it as DVM2, and change the GPIB address in the second VI to match that of the second voltmeter. Because each VI can execute asynchronously, the wait periods can overlap as shown below, making the overall operation of your application more efficient.



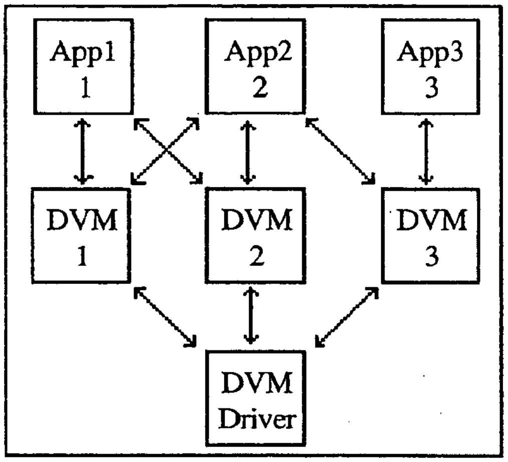
This approach is acceptable, but it uses extra memory and disk space because you duplicated the original DVM VI front panel, block diagram, machine code, and data space.

An alternate approach is to put another control on the front panel of the original DVM for specifying the GPIB address of the voltmeter. Now you can use a single VI to control both voltmeters. However, because the normal execution behavior of a subVI is serially reusable, the execution time line will look something like the one below.



You can restore the efficiency of the first case without duplicating your VI by making the DVM VI reentrant. Multiple parallel calls to the DVM subVI execute in parallel. However, when you make a VI reentrant, you must ensure that all the calls you make in parallel can legitimately occur in parallel. For example, making two calls in parallel to the same voltmeter corrupts the communication.

The diagram below shows the recommended architecture for managing multiple identical physical resources that you can use in parallel.

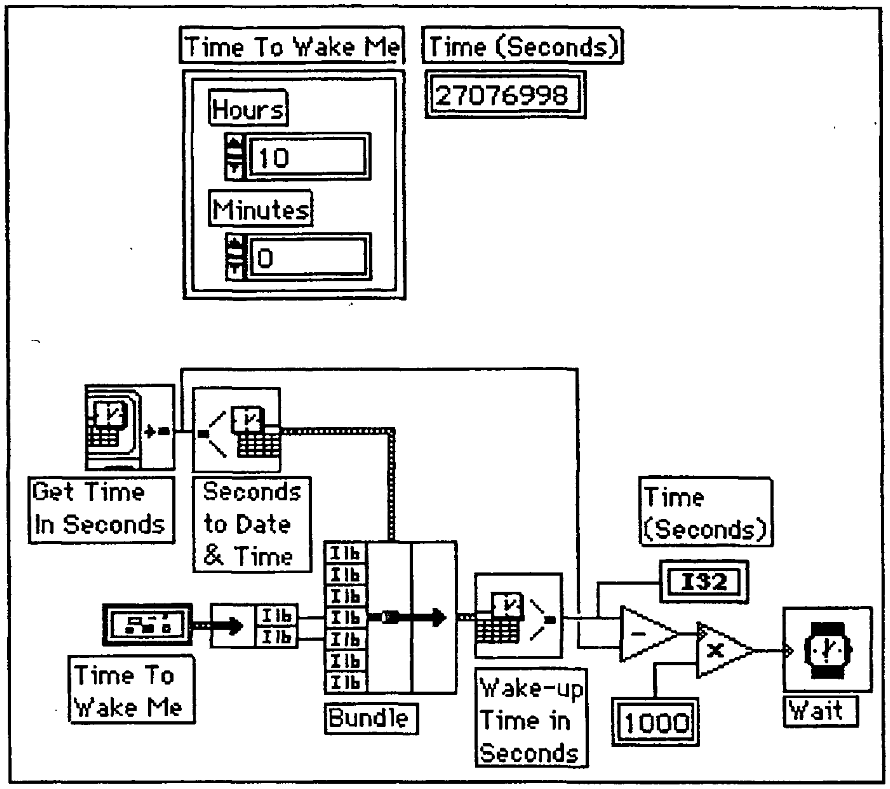


The three VIs at the top, App1 through App3, are top-level applications. One application might create a voltage versus frequency response plot of a unit under test. Another might perform inspection of resistors. The third might calibrate another instrument. At the bottom of the hierarchy is a reentrant instrument driver VI, DVM Driver. In the middle are three non-reentrant subVIs, DVM 1 through DVM 3, whose purpose might be to serialize requests for each instrument. Each of these subVIs contains a call to DVM Driver and passes the instrument's address and front panel settings. This architecture prevents parallel use of one voltmeter by multiple applications but permits parallel executions of separate voltmeters by those applications.

A VI That Waits

zzZZ

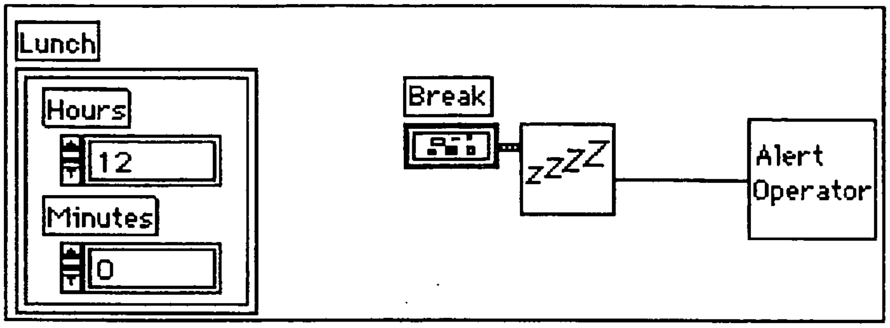
The following reentrant example discusses a VI, called Snooze, which takes hours and minutes as input and waits until that time arrives.



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.

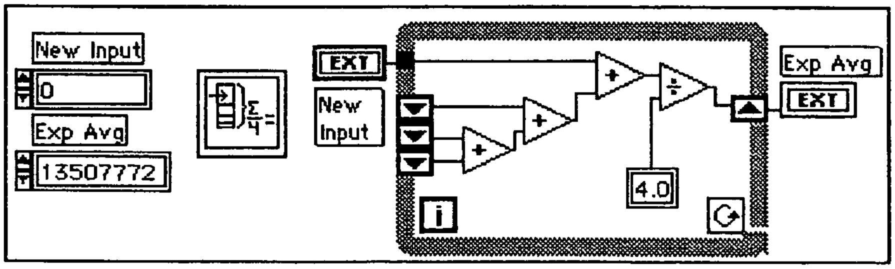
Lunch and Break are two VIs that use Snooze as a subVI. The Lunch VI, whose front panel and block diagram are shown below, waits until noon and pops up a panel reminding the operator to go to lunch. The Break VI pops up a panel reminding the operator to go on break at 10:00 a.m. The Break VI is identical 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.

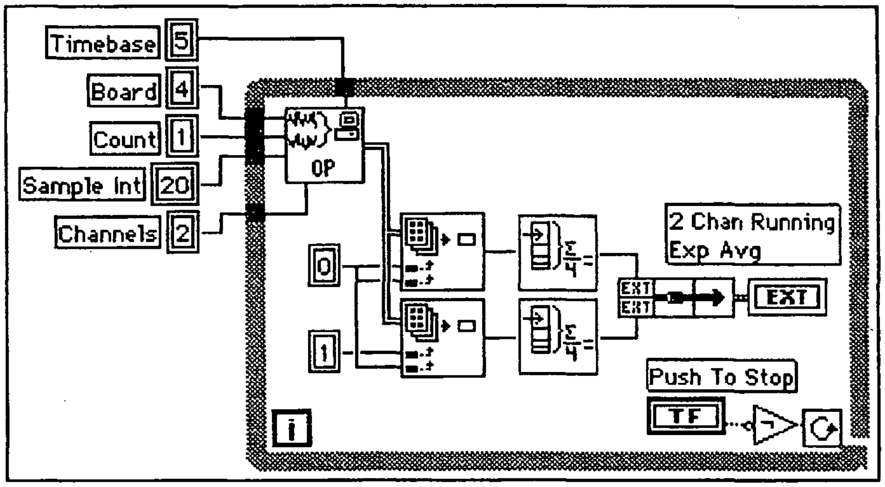


A Storage VI Not Meant to Share Its Data

A third 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 your tutorial manual for more information on uninitialized shift registers.



Next, suppose that 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.



Appendices

Appendices

Appendix

A

Data Storage Formats

This appendix discusses the formats in which LabVIEW saves data. This information is most useful to advanced LabVIEW users, such as those using code interface nodes and those reading from or writing to files used by the file I/O functions. This appendix 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.

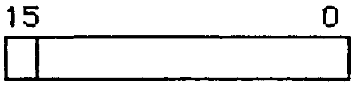
Data Representation in Memory

The following sections describe the memory representation of LabVIEW front panel objects.

Data Formats of LabVIEW Front Panel Controls and Indicators

Boolean

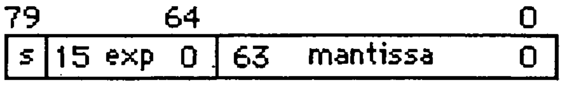
Booleans are 16-bit integers. The most significant bit contains the Boolean value. If bit 15 is set to 1, then the value of the control is TRUE; if it is set to 0, then the value is FALSE.



Numeric

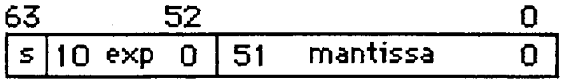
Extended

Extended-precision floating-point numbers have 80-bit format (80287 extended-precision format).



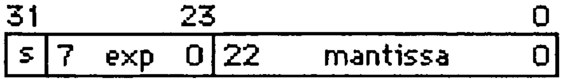
Double

Double-precision floating-point numbers have 64-bit IEEE double-precision format (LabVIEW default).



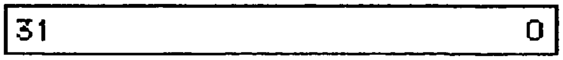
Single

Single-precision floating-point numbers have 32-bit IEEE single-precision format.



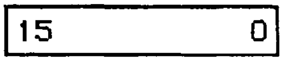
Long Integer

Long integer numbers have 32-bit format, signed or unsigned.



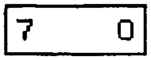
Word Integer

Word integer numbers have 16-bit format, signed or unsigned.



Byte Integer

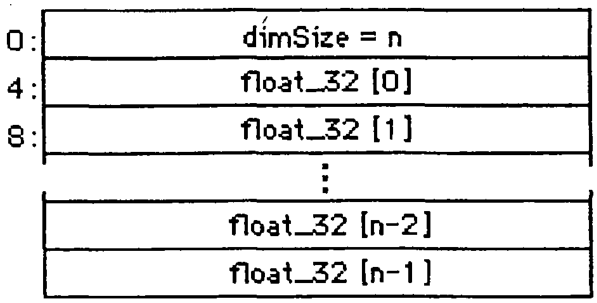
Byte integer numbers have 8-bit format, signed or unsigned.



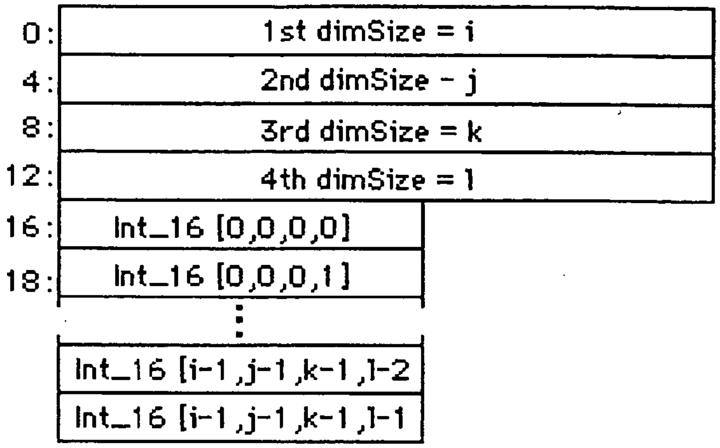
Arrays

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 Consideration* section of Chapter 2, *Parameter Passing*, of the *Code Interface Reference Manual* for further information.

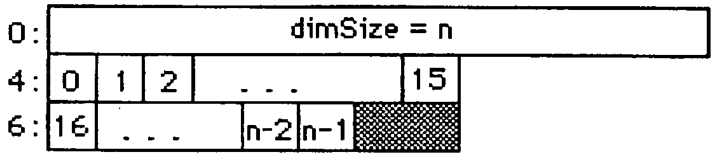
LabVIEW stores the size of each dimension of an array in long integers, followed by the data. The example that follows shows a one-dimensional array of single-precision floating-point numbers. The decimal numbers to the left represent the offsets of the location in memory where each array begins.



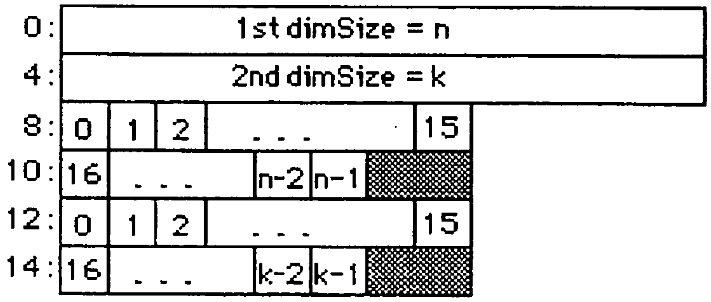
This illustration shows a four-dimensional array of word integers.



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}).

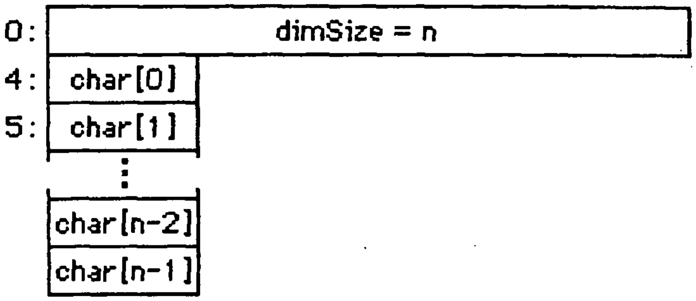


This illustration shows an example of a two-dimensional Boolean array. LabVIEW stores each dimension's 0th element in a new word integer and ignores unused bits from previous dimensions.



Strings

LabVIEW stores strings as if they were one-dimensional arrays of byte integers (8-bit characters).



Handles

A *handle* is pointer to a pointer to a relocatable block of memory. A handle points to user-defined data only; LabVIEW does not recognize what is in the block of memory to which the handle points. A handle is particularly useful for passing a block of data by reference between code interface nodes.

Paths

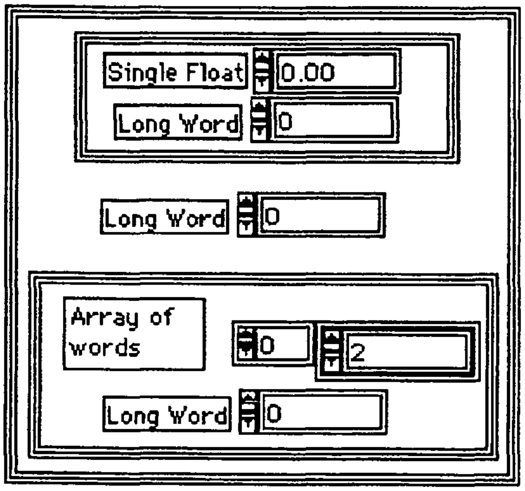
LabVIEW stores the path type and number of path components in word integers, followed immediately by the path components. The path type is 0 for an absolute path and 1 for a relative path. Any other value of path type indicates that the path is invalid. Each path component is a Pascal string (P-string) in which the first byte is the length, in bytes, of the P-string (not including the length byte).

Clusters

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. The illustration below shows 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. For more information, see Chapter 8, *Array and Cluster Controls and Indicators.*

0:	SGL float
4:	EXT float
14:	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.



Single Float 0.00
Long Word 0
Long Word 0
Array of words 0 2
Long Word 0

0:	32-bit float
4:	32-bit float
8:	32-bit init
12:	Handle to Array
16:	32-bit init

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 typecode. 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 typecode 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 typecodes are not.

The typecode encodes the actual type information, such as single-precision or extended-precision floating-point number, as listed in the following table. These typecode values may change in future versions of LabVIEW. The **xx** in the type descriptor columns represent reserved LabVIEW values and should be ignored.

Data Types

The following tables list scalar numeric and non-numeric data types, the type codes, and type descriptors.

Table A-1. Scalar Numeric Data Types

Data Type	**Type Code**	**Type Descriptor**
Byte Integer	0x01	0004 *xx*01
Word Integer	0x02	0004 *xx*02
Long Integer	0x03	0004 *xx*03
Unsigned Byte Integer	0x05	0004 *xx*05
Unsigned Word Integer	0x06	0004 *xx*06
Unsigned Long Integer	0x07	0004 *xx*07
Single-Precision Floating-Point Number	0x09	0004 *xx*09
Double-Precision Floating-Point Number	0x0A	0004 *xx*0A
Extended-Precision Floating-Point Number	0x0B	0004 *xx*0B

Table A-1. Scalar Numeric Data Types (Continued)

Data Type	Type Code	Type Descriptor
Single-Precision Complex Floating-Point Number	0x0C	0004 *xx*0C
Double-Precision Complex Floating-Point Number	0x0D	0004 *xx*0D
Extended-Precision Complex Floating-Point Number	0x0E	0004 *xx*0E
Enumerated Byte Integer	0x15	<nn> *xx*15 <k> <k pstrs>
Enumerated Word Integer	0x16	<nn> *xx*16 <k> <k pstrs>
Enumerated Long Integer	0x17	<nn> *xx*17 <k> <k pstrs>
Single-Precision Physical Quantity	0x19	<nn> *xx*19 <k> <k base-exp>
Double-Precision Physical Quantity	0x1A	<nn> *xx*1A <k> <k base-exp>
Extended-Precision Physical Quantity	0x1B	<nn> *xx*1B <k> <k base-exp>
Single-Precision Complex Physical Quantity	0x1C	<nn> *xx*1C <k> <k base-exp>
Double-Precision Complex Physical Quantity	0x1D	<nn> *xx*1D <k> <k base-exp>
Extended-Precision Complex Physical Quantity	0x1E	<nn> *xx*1E <k> <k base-exp>

Table A-2. Non-Numeric Data Types

Data Type	Type Code	Type Descriptor
Boolean	0x20	0004 *xx*20
String	0x30	0004 *xx*30
Handle	0x31	0006 *xx*31 <kind>
Path	0x32	0004 *xx*32
Pict	0x33	0004 *xx*33
Array	0x40	<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 typecode. 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 a ASCII space.

```
0x16 0x15 0003 02a m02 fm 09f m" " st er eo
```

0x16 indicates 22 bytes total. 0x15 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.

```
0x0E 0x1A 0002 0003 0001 0002 FFFF
```

0x0E indicates 14 bytes total. 0x1A indicates that these are double-precision width units. 0002 indicates two base-exponent pairs. 0003 denotes the meters base index. 0001 is the exponent of meters. 0002 denotes the seconds base index. FFFF is the exponent of / s)

Note: ***All physical quantities are always stored internally in terms of base units, regardless of what unit they are displayed as. Table 5-2, Base Units, shows the nine bases which are represented by indices*** 0-8 ***for*** radians-candela.

Array

The typecode for an array is 0X40. Immediately after the typecode 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 and 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

The typecode for a cluster is 0X50. Immediately after the typecode is a word containing the number of items in the cluster. After this word is the type descriptor for each element in a *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 0050z 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.

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 wanted to write the cluster data to disk, you would 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. It stores numeric data in "big endian" form, and it stores extended precision floating-point numbers as 16-byte quantities. LabVIEW packs byte-sized data, and aligns all other data to word boundaries.

The Flatten to String and Unflatten from String block diagram functions (described in the *LabVIEW Function Reference Manual*) use the internal flatten and unflatten functions.

The LabVIEW file form for extended-precision numbers is based on the SPARC extended-precision form, which is a 16 bit biased exponent, the same as for a Macintosh, followed by a mantissa which is 112 bits long. This mantissa is not just a Macintosh mantissa with an extra unsigned long integer attached. 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. LabVIEW therefore shifts the Macintosh and PC mantissa left by one bit and saves it. To load the mantissa, LabVIEW shifts it right by one and sets the explicit bit to a one. In the rare case of very small denormalized numbers a special case handles whether to set the explicit bit or not.

Scalars

The flattened form of any numeric type, as well as the Boolean type, contains only the data. 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 π is `0X 40 09 21 FB 54 44 2D 18`. A Boolean True is `8xxx`, `9xxx`, ..., `Exxx`, or `Fxxx`.

Strings, Handles, and Paths

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`. After the flattened data, there may be an additional pad byte to align subsequent data to a word boundary. So, for instance, a string type with value `ABC` would be flattened to `0000 0003 4142 4300`.

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.

Arrays

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. Also if necessary, it pads arrays with an additional byte to align subsequent data to a word boundary. 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.
```

Notice that there is no padding after the first 8-bit integer array element, even though it is of odd-size, because LabVIEW does not pad between byte-sized data.

The following is an example of a flattened one-dimensional array of strings.

```
{"ABC", "DEFG"} is stored as 0000 0002 0000 0003
4142 4300 0000 0004 4445 4647.
```

Notice that there is one byte of padding after the first string array element. Its purpose is to align the start of the subsequent string array element to a word boundary, because the first string is of odd-size but not byte size. In fact, no flattened string could be byte-sized since a flattened string begins with a long integer (4-byte) length.

Finally, 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, as explained in the *Arrays* section.

Clusters

If necessary, LabVIEW pads clusters with an additional byte to align subsequent data to a word boundary.

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 0004.

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.

Appendix B

Error Codes

This appendix contains tables of numeric error codes drawn from the entire LabVIEW documentation set.

You can connect error handler VIs to the error status terminals of other VIs to return a text-based description of the error, if one occurs. Error handler VIs can also display a dialog box with a description of the error message, and with buttons that allow the user to stop or continue execution. See Chapter 18, *Error Handler VIs*, for more information on using error handlers.

The tables are arranged roughly in ascending order, from negative values to positive. Tables with negative number values are arranged from the smallest absolute value to the largest absolute value.

Error Code Range	**Table**
-20001 to - 20060	Analysis VI Error Codes
-10001 to - 10848	Data Acquisition VI Error Codes
-60 to -81	Data Acquisition Configuration Utility Error Codes
* 0 to 43	LabVIEW Function Error Codes
* 0 to 65	GPIB Error Codes
* 54 to 66	TCP/IP Error Codes
14001 to 14020	DDE Error Codes

* These tables contain some error codes with overlapping numerical values but different meanings, depending on the source of the error.

Table B-1. Analysis VI Error Codes

Code	Name	Description
0	NoErr	No error; the call was successful.
-20001	OutOfMemErr	There is not enough memory left to perform the specified routine.
-20002	EqSamplesErr	The input sequences must be the same size.
-20003	SamplesGTZeroErr	The number of samples must be greater than zero.
-20004	SamplesGEZeroErr	The number of samples must be greater than or equal to zero.
-20005	SamplesGEOneErr	The number of samples must be greater than or equal to one.
-20006	SamplesGETwoErr	The number of samples must be greater than or equal to two.
-20007	SamplesGEThreeErr	The number of samples must be greater than or equal to three.
-20008	ArraySizeErr	The input arrays do not meet the specified conditions.
-20009	PowerOfTwoErr	The size of the input array must be a valid power of two: size = 2^m, $0<m<23$.
-20010	MaxXformSizeErr	The maximum transform size has been exceeded.
-20011	DutyCycleErr	The duty cycle must meet the condition: 0≤duty cycle≤100.
-20012	CyclesErr	The number of cycles must be greater than zero and less than or equal to the number of samples.
-20013	WidthLTSamplesErr	The width must meet the condition: 0<width<samples.
-20014	DelayWidthErr	The following condition must be met: 0≤(delay+width)<samples.
-20015	DtGEZeroErr	The dt parameter must be greater than or equal to zero.
-20016	DtGTZeroErr	The dt parameter must be greater than zero.
-20017	IndexLTSamplesErr	The index must meet the condition: 0≤index<samples.
-20018	IndexLengthErr	The following condition must be met: 0≤(index+length)<samples.
-20019	UpperGELowerErr	The upper value must be greater than or equal to the lower value.

Table B-1. Analysis VI Error Codes (Continued)

Code	Name	Description
-20020	NyquistErr	The cutoff frequency, f_c, must meet the condition: $0 \le f_c \le \frac{f_s}{2}$
-20021	OrderGTZeroErr	The order must be greater than zero.
-20022	DecFactErr	The decimating factor must meet the condition: 0<decimating≤samples.
-20023	BandSpecErr	The following condition must be met: $0 \le f_{flow} \le f_{high} \le \frac{f_s}{2}$
-20024	RippleGTZeroErr	The ripple amplitude must be greater than zero.
-20025	AttenGTZeroErr	The attenuation must be greater than zero.
-20026	WidthGTZeroErr	The width must be greater than zero.
-20027	FinalGTZeroErr	The final value must be greater than zero
-20028	AttenGTRippleErr	The attenuation must be greater than the ripple amplitude.
-20029	StepSizeErr	The step-size parameter μ must meet the condition: 0 ≤μ≤0.1.
-20030	LeakErr	The leakage coefficient must meet the condition: 0≤Leak≤μ.
-20031	EqRplDesignErr	The filter cannot be designed with the specified input parameters.
-20032	RankErr	The rank of the filter must meet the condition: 1≤(2 rank + 1)≤size.
-20033	EvenSizeErr	The number of coefficients must be odd for this filter.
-20034	OddSizeErr	The number of coefficients must be even for this filter.
-20035	StdDevErr	The standard deviation is zero. The normalization is impossible.
-20036	MixedSignErr	The elements of the second array input must be nonzero and either all positive or all negative.
-20037	SizeGTOrderErr	The array size must be greater than the order.
-20038	IntervalsErr	The number of intervals must be greater than zero.
-20039	MatrixMulErr	The specified matrix multiplication must be a square matrix.
-20040	SquareMatrixErr	The input matrix must be a square matrix.

Table B-1. Analysis VI Error Codes (Continued)

Code	Name	Description
-20041	SingularMatrixErr	The input matrix is singular. The system of equations cannot be solved.
-20042	LevelsErr	The numbers of levels is out of range.
-20043	FactorErr	The level of factors is out of range for some data.
-20044	ObservationsErr	Zero observations were made at some level of a factor.
-20045	DataErr	The total number of data points must be equal to the product of the levels for each factor and the observations per cell.
-20046	OverflowErr	There is an overflow in the calculated F-value.
-20047	BalanceErr	The data is unbalanced.
-20048	ModelErr	The Random Effect model was requested when the Fixed Effect model was required.
-20049	DistinctErr	The x values must be distinct.
-20050	PoleErr	The interpolating function has a pole at the requested value.
-20051	ColumnErr	The values in the first column in the X matrix must all be ones.
-20052	FreedomErr	The degrees of freedom are invalid.
-20053	ProbabilityErr	The probability must meet the condition: 0 < p < 1.
-20054	InvProbErr	The probability must meet the condition: 0 ≤ p < 1.
-20055	CategoryErr	The number of categories or samples is invalid.
-20056	TableErr	The contingency table contains a negative number.
-20057	BetaFuncErr	The parameter to the beta function must meet the condition: 0 < p < 1
-20058	DimensionErr	The number of dimensions or dependent variables is invalid.
-20059	NegativeNumErr	The input value must not be negative.
-20060	InvalidSelectionErr	One of the input selections is invalid.

Table B-2. Data Acquisition VI Error Codes

Code	Name	Description
-10001	syntaxErr	An error was detected in the input string; the arrangement or ordering of the characters in the string is not consistent with the expected ordering.
-10002	semanticsErr	An error was detected in the input string; the syntax of the string is correct, but certain values specified in the string are inconsistent with other values specified in the string.
-10003	invalidValueErr	The value of a numeric parameter is invalid.
-10004	valueConflictErr	The value of a numeric parameter is inconsistent with another parameter, and the combination is therefore invalid.
-10005	badDeviceErr	The device parameter is invalid.
-10006	badLineErr	The line parameter is invalid.
-10007	badChanErr	A channel is out of range for the board type or input configuration, the combination of channels is not allowed, or you must reverse the scan order so that channel 0 is last.
-10008	badGroupErr	The group parameter is invalid.
-10009	badCounterErr	The counter parameter is invalid.
-10010	badCountErr	The count parameter is too small or too large for the specified counter.
-10011	badIntervalErr	The interval parameter is too small or too large for the associated counter or I/O channel.
-10012	badRangeErr	The analog input or analog output voltage range is invalid for the specified channel.
-10013	badErrorCodeErr	The driver returned an unrecognized or unlisted error code.
-10014	groupTooLargeErr	The group size is too large for the board.
-10015	badTimeLimitErr	The time limit parameter is invalid.
-10016	badReadCountErr	The read count parameter is invalid.
-10017	badReadModeErr	The read mode parameter is invalid.
-10018	badReadOffsetErr	The offset is unreachable.
-10019	badClkFrequencyErr	The frequency parameter is invalid.
-10020	badTimebaseErr	The timebase parameter is invalid.
-10021	badLimitsErr	The limits are beyond the range of the board.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10022	badWriteCountErr	Your data array contains an incomplete update or you are trying to write past the end of the internal buffer or your output operation is continuous and the length of your array is not a multiple of one half of the internal buffer size.
-10023	badWriteModeErr	The write mode is out of range or is disallowed.
-10024	badWriteOffsetErr	The write offset plus the write mark is greater than the internal buffer size or it must be set to 0
-10025	limitsOutOfRangeErr	The voltage limits are out of range for this board in the current configuration. Alternate limits were selected.
-10026	badInputBufferSpecification	The input buffer specification is invalid. This error results if, for example, you try to configure a multiple-buffer acquisition for a board that does not support multiple-buffer acquisition.
-10027	badDAQEventErr	For DAQEvents 0 and 1, general value A must be greater than 0 and less than the internal buffer size. If DMA is used for DAEvent 1, general value A must divide the internal buffer evenly. If the TIO-10 is used for DAQEvent 4, general value A must be 1 or 2
-10080	badGainErr	The gain parameter is invalid.
-10081	badPretrigCountErr	The pretrigger sample count is invalid.
-10082	badPosttrigCountErr	The posttrigger sample count is invalid.
-10083	badTrigModeErr	The trigger mode is invalid.
-10084	badTrigCountErr	The trigger count is invalid.
-10085	badTrigRangeErr	The trigger range or trigger hysteresis window is invalid.
-10086	badExtRefErr	The external reference value is invalid.
-10087	badTrigTypeErr	The trigger type parameter is invalid.
-10088	badTrigLevelErr	The trigger level parameter is invalid.
-10089	badTotalCountErr	The total count specified is inconsistent with the buffer configuration and pretrigger scan count or with the board type.
-10090	badRPGErr	The individual range, polarity, and gain settings are valid but the combination specified is not allowed for this board.
-10091	badIterationsErr	The analog output buffer iterations count is not allowed. It must be 0 (for indefinite iterations) or 1

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10100	badPortWidthErr	The requested digital port width is not a multiple of the hardware port width.
-10240	noDriverErr	The driver interface could not locate or open the driver.
-10241	oldDriverErr	The driver is out-of-date.
-10242	functionNotFoundErr	The specified function is not located in the driver.
-10243	configFileErr	The driver could not locate or open the configuration file, or the format of the configuration file is not compatible with the currently installed driver.
-10244	deviceInitErr	The driver encountered a hardware-initialization error while attempting to configure the specified device.
-10245	osInitErr	The driver encountered an operating system error while attempting to perform an operation, or the operating system does not support an operation performed by the driver.
-10246	communicationsErr	The driver is unable to communicate with the specified external device.
-10247	cmosConfigErr	The CMOS configuration memory for the computer is empty or invalid, or the configuration specified does not agree with the current configuration of the computer.
-10248	dupAddressErr	The base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.
-10249	intConfigErr	The interrupt configuration is incorrect given the capabilities of the computer or device.
-10250	dupIntErr	The interrupt levels for two or more devices are the same.
-10251	dmaConfigErr	The DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
-10252	dupDMAErr	The DMA channels for two or more devices are the same.
-10340	noConnectErr	No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
-10341	badConnectErr	The RTSI signal/line cannot be connected as specified.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10342	multConnectErr	The specified RTSI signal is already being driven by a RTSI line, or the specified RTSI line is already being driven by a RTSI signal.
-10343	SCXIConfigErr	The specified SCXI configuration parameters are invalid, or the function cannot be executed given the current SCXI configuration.
-10360	DSPInitErr	The DSP driver was unable to load the kernel for its operating system.
-10370	badScanListError	The scan list is invalid. This error can result if, for example, you mix AMUX-64T channels and onboard channels, or if you scan multiplexed SCXI channels out of order.
-10400	userOwnedRsrcErr	The specified resource is owned by the user and cannot be accessed or modified by the driver.
-10401	unknownDeviceErr	The specified device is not a National Instruments product, or the driver does not support the device (for example, the driver was released before the device was supported).
-10402	deviceNotFoundErr	No device is located in the specified slot or at the specified address.
-10403	deviceSupportErr	The specified device does not support the requested action (the driver recognizes the device, but the action is inappropriate for the device).
-10404	noLineAvailErr	No line is available.
-10405	noChanAvailErr	No channel is available.
-10406	noGroupAvailErr	No group is available.
-10407	lineBusyErr	The specified line is in use.
-10408	chanBusyErr	The specified channel is in use.
-10409	groupBusyErr	The specified group is in use.
-10410	relatedLCGBusyErr	A related line, channel, or group is in use; if the driver configures the specified line, channel, or group, the configuration, data, or handshaking lines for the related line, channel, or group will be disturbed.
-10411	counterBusyErr	The specified counter is in use.
-10412	noGroupAssignErr	No group is assigned, or the specified line or channel cannot be assigned to a group.
-10413	groupAssignErr	A group is already assigned, or the specified line or channel is already assigned to a group.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10440	sysOwnedRsrcErr	The specified resource is owned by the driver and cannot be accessed or modified by the user.
-10441	memConfigErr	No memory is configured to support the current data transfer mode, or the configured memory does not support the current data transfer mode. (If block transfers are in use, the memory must be capable of performing block transfers.)
-10442	memDisabledErr	The specified memory is disabled or is unavailable given the current addressing mode.
-10443	memAlignmentErr	The transfer buffer is not aligned properly for the current data transfer mode. For example, the memory buffer is at an odd address, is not aligned to a 32-bit boundary, is not aligned to a 512-bit boundary, and so on. Alternatively, the driver is unable to align the buffer because the buffer is too small.
-10444	memFullErr	No more system memory is available on the heap, or no more memory is available on the device.
-10445	memLockErr	The transfer buffer cannot be locked into physical memory.
-10446	memPageErr	The transfer buffer contains a page break; system resources may require reprogramming when the page break is encountered.
-10447	memPageLockErr	The operating environment is unable to grant a page lock.
-10448	stackMemErr	The driver is unable to continue parsing a string input due to stack limitations.
-10449	cacheMemErr	A cache-related error occurred, or caching is not supported in the current mode.
-10450	physicalMemErr	A hardware error occurred in physical memory, or no memory is located at the specified address.
-10451	virtualMemErr	The driver is unable to make the transfer buffer contiguous in virtual memory and therefore cannot lock the buffer into physical memory; thus, you cannot use the buffer for DMA transfers.
-10452	noIntAvailErr	No interrupt level is available for use.
-10453	intInUseErr	The specified interrupt level is already in use by another device.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10454	noDMACErr	No DMA controller is available in the system.
-10455	noDMAAvailErr	No DMA channel is available for use.
-10456	DMAInUseErr	The specified DMA channel is already in use by another device.
-10457	badDMAGroupErr	DMA cannot be configured for the specified group because it is too small, too large, or misaligned. Consult the user manual for the device in question to determine group ramifications with respect to DMA.
-10459	DLLInterfaceErr	The DLL could not be called due to an interface error.
-10460	interfaceInteractionErr	You have attempted to mix LabVIEW 2.2 VIs and LabVIEW 3.0 VIs. You may run an application consisting only of 2.2 VIs, then run the 2.2 Board Reset VI, before you can run any 3.0 VIs. You may run an application consisting of only 3.0 VIs, then run the 3.0 Device Reset VI, before you can run any 2.2 VIs.
-10560	invalidDSPhandleError	The DSP handle input to the VI is not a valid handle.
-10600	noSetupErr	No setup operation has been performed for the specified resources.
-10601	multSetupErr	The specified resources have already been configured by a setup operation.
-10602	noWriteErr	No output data has been written into the transfer buffer.
-10603	groupWriteErr	The output data associated with a group must be for a single channel or must be for consecutive channels.
-10604	activeWriteErr	Once data generation has started, only the transfer buffers originally written to can be updated. If DMA is active and a single transfer buffer contains interleaved channel-data, new data must be provided for all output channels currently using the DMA channel.
-10605	endWriteErr	No data was written to the transfer buffer because the final data block has already been loaded.
-10606	notArmedErr	The specified resource is not armed.
-10607	armedErr	The specified resource is already armed.
-10608	noTransferInProgErr	No transfer is in progress for the specified resource.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10609	transferInProgErr	A transfer is already in progress for the specified resource.
-10610	transferPauseErr	A single output channel in a group cannot be paused if the output data for the group is interleaved.
-10611	badDirOnSomeLinesErr	Some of the lines in the specified channel are not configured for the transfer direction specified. For a write transfer, some lines were configured for input. For a read transfer, some lines were configured for output.
-10612	badLineDirErr	The specified line does not support the specified transfer direction.
-10613	badChanDirErr	The specified channel does not support the specified transfer direction.
-10614	badGroupDirErr	The specified group does not support the specified transfer direction.
-10615	masterClkErr	The clock configuration for the clock master is invalid.
-10616	slaveClkErr	The clock configuration for the clock slave is invalid.
-10617	noClkSrcErr	No source signal has been assigned to the clock resource.
-10618	badClkSrcErr	The specified source signal cannot be assigned to the clock resource.
-10619	multClkSrcErr	A source signal has already been assigned to the clock resource.
-10620	noTrigErr	No trigger signal has been assigned to the trigger resource.
-10621	badTrigErr	The specified trigger signal cannot be assigned to the trigger resource.
-10622	preTrigErr	The pretrigger mode is not supported or is not available in the current configuration, or no pretrigger source has been assigned.
-10623	postTrigErr	No posttrigger source has been assigned.
-10624	delayTrigErr	The delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
-10625	masterTrigErr	The trigger configuration for the trigger master is invalid.
-10626	slaveTrigErr	The trigger configuration for the trigger slave is invalid.
-10627	noTrigDrvErr	No signal has been assigned to the trigger resource.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10628	multTrigDrvErr	A signal has already been assigned to the trigger resource.
-10629	invalidOpModeErr	The specified operating mode is invalid, or the resources have not been configured for the specified operating mode.
-10630	invalidReadErr	An attempt was made to read 0 bytes from the transfer buffer, or an attempt was made to read past the end of the transfer buffer.
-10631	noInfiniteModeErr	Continuous input or output transfers are not allowed in the current operating mode.
-10632	someInputsIgnoredErr	Certain inputs were ignored because they are not relevant in the current operating mode.
-10633	invalidRegenModeError	This board does not support the specified analog output regeneration mode.
-10680	badChanGainErr	All channels must have an identical setting for this board.
-10681	badChanRangeErr	All channels of this board must have the same range.
-10682	badChanPolarityErr	All channels of this board must have the same polarity.
-10683	badChanCouplingErr	All channels of this board must have the same coupling.
-10684	badChanInputModeErr	All channels of this board must have the same input range.
-10685	clkExceedsBrdsMaxConvRate	The clock rate selected exceeds the recommended maximum rate for this board.
-10686	scanListInvalidErr	A configuration change has invalidated the scan list.
-10687	bufferInvalidErr	A configuration change has invalidated the allocated buffer.
-10688	noTrigEnabledErr	The total number of scans and pretrigger scans implies that a trigger start is intended, but no trigger is enabled.
-10689	digitalTrigBErr	Digital trigger B is illegal for the total scans and pretrigger scans specified.
-10690	digitalTrigAandBErr	This board does not allow digital triggers A and B to be enabled at the same time.
-10691	extConvRestrictionErr	This board does not allow an external sample clock with an external scan clock, start trigger, or stop trigger.
-10692	chanClockDisabledErr	Cannot start the acquisition because the channel clock is disabled.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10693	extScanClockError	Cannot use an external scan clock when performing a single scan of a single channel.
-10694	unsafeSamplingFreqError	The sampling frequency exceeds the safe maximum rate for the ADC, gains, and filters you are using.
-10695	DMANotAllowedErr	You must use interrupts. DMA is not allowed.
-10740	SCXITrackHoldErr	A signal has already been assigned to the SCXI track-and-hold trigger line, or a control call was inappropriate because the specified module is not configured for one-channel operation.
-10800	timeOutErr	The operation could not complete within the time limit.
-10801	calibrationErr	An error occurred during the calibration process.
-10802	dataNotAvailErr	The requested amount of data has not yet been acquired, or the acquisition has completed and no more data is available to read.
-10803	transferStoppedErr	The transfer has been stopped to prevent regeneration of output data.
-10804	earlyStopErr	The transfer stopped prior to reaching the end of the transfer buffer.
-10805	overRunErr	The clock source for the input transfer is faster than the maximum input-clock rate; the integrity of the data has been compromised. Alternatively, the clock source for the output transfer is faster than the maximum output-clock rate; a data point was generated more than once since the update occurred before new data was available.
-10806	noTrigFoundErr	No trigger value was found in the input transfer buffer.
-10807	earlyTrigErr	The trigger occurred before sufficient pretrigger data was acquired.
-10840	softwareErr	The contents or the location of the driver file was changed between accesses to the driver.
-10841	firmwareErr	The firmware does not support the specified operation, or the firmware operation could not complete due to a data-integrity problem.

Table B-2. Data Acquisition VI Error Codes (Continued)

Code	Name	Description
-10842	hardwareErr	The hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
-10843	underFlowErr	The update rate exceeds your system's capacity to supply data to the output channel.
-10844	underWriteErr	At the time of the update for the device-resident memory, insufficient data was present in the output transfer buffer to complete the update.
-10845	overFlowErr	At the time of the update clock for the input channel, the device-resident memory was unable to accept additional data–one or more data points may have been lost.
-10846	overWriteErr	New data was written into the input transfer buffer before the old data was retrieved.
-10847	dmaChainingErr	New buffer information was not available at the time of the DMA chaining interrupt; DMA transfers will terminate at the end of the currently active transfer buffer.
-10848	noDMACountAvailErr	The driver could not obtain a valid reading from the transfer-count register in the DMA controller.

Table B-3. Data Acquisition Configuration Utility Error Codes

Code	Name	Description
-60	notOurBrdErr	The board in the specified slot is not an MC Series, AT Series, EISA Series, or Lab-PC board.
-61	badBrdNumErr	The board parameter is out of range.
-62	badGainErr	The gain parameter is out of range.
-63	badChanErr	The channel parameter is out of range.
-64	noSupportErr	Function cannot be executed by the specified board.
-65	badPortErr	The port parameter is out of range or the port is busy.
-66	badOutPortErr	The specified port has not been configured as an output port.
-67	noLatchModeErr	The specified port has not been configured for handshaking.

Table B-3. Data Acquisition Configuration Utility Error Codes (Continued)

Code	Name	Description
-69	badInputValErr	One or more input parameters are out of range.
-70	timeOutErr	A/D conversion did not complete or timeout period has expired.
-71	outOfRangeErr	Scaled input value is out of range.
-72	daqInProgErr	Data acquisition is in progress; therefore, call was not executed.
-75	overFlowErr	A/D FIFO memory has overflowed as a result of a DAQ or SCAN operation.
-76	overRunErr	Minimum sample interval has been exceeded as a result of a DAQ or SCAN operation.
-81	portAssignToGrp	The specified port is currently assigned to a group and can be accessed only through digital group calls until unassigned.

Table B-4. LabVIEW Function Error Codes

Code	Name	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.

Table B-4. LabVIEW Function Error Codes (Continued)

Code	Name	Description
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.

Table B-4. LabVIEW Function Error Codes (Continued)

Code	Name	Description
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.

Table B-5. GPIB Error Codes

Code	Name	Description
0	EDVR	Error connecting to driver.
1	ECIC	Command requires GPIB Controller to be CIC.
2	ENOL	Write detected no Listeners.
3	EADR	GPIB Controller not addressed correctly.
4	EARG	Invalid argument or arguments.
5	ESAC	Command requires GPIB Controller to be SC.
6	EABO	I/O operation aborted.
7	ENEB	Non-existent board.
8	EDMA	DMA Hardware error detected.
9	EBTO	DMA hardware µP bus timeout.
11	ECAP	No capability.
12	EFSO	File system operation error.
13	EOWN	Shareable board exclusively owned.
14	EBUS	GPIB bus error.
15	ESTB	Serial poll byte queue overflow.
16	ESRQ	SRQ stuck on.
17	ECMD	Unrecognized command.
19	EBNP	Board not present.

Table B-5. GPIB Error Codes (Continued)

Code	Name	Description
20	ETAB	Table error.
30	NADDR	No GPIB address input.
31	NSTRG	No string input (write).
32	NCNT	No count input (read).
61	EPAR	Serial port parity error.
62	EORN	Serial port overrun error.
63	EOFL	Serial port receive buffer overflow.
64	EFRM	Serial port framing error.
65	SPTMO	Serial port timeout, bytes not received at serial port.

Table B-6. TCP/IP Error Codes

Code	Name	Description
54	ncBadAddress	The net address was ill formed
55	ncInProgressErr	Operation is in progress.
56	ncTimeOutErr	Operation exceeded the user-specified time limit.
57	ncBusyErr	The connection was busy.
58	ncNotSupportedErr	Function not supported.
59	ncNetErr	The network is down, unreachable, or has been reset.
60	ncAddrInUseErr	The specified address is currently in use.
61	ncSysOutOfMemErr	System could not allocate necessary memory.
62	ncSysConAbortedErr	System caused connection to be aborted.
63	ncConRefusedErr	Connection is not established.
65	ncAlreadyConnectedErr	Connection is already established.
66	ncConClosedErr	Connection was closed by peer.

Table B-7. DDE Error Codes

Code	Name	Description
00000		No error.
14001	DDE_INVALID_REFNUM	Invalid refnum.
14002	DDE_INVALID_STRING	Invalid string.
14003	DDEML_ADVACKTIMEOUT	Request for a synchronous advise transaction has timed out.
14004	DDEML_BUSY	Response set the `DDE_FBUSY` bit.
14005	DDEML_DATAACKTIMEOUT	Request for a synchronous data transaction has timed out.
14006	DDEML_DDL_NOT_INITIALIZED	DDEML called without first calling `DdeInitialize`, or was passed an invalid instance identifier.
14007	DDEML_DLL_USAGE	A monitor or client-only application has attempted a DDE transaction.
14008	DDEML_EXECACKTIMEOUT	Request for a synchronous execute transaction has timed out.
14009	DDEML_INVALIDPARAMETER	Parameter not validated by the DDML.
14010	DDEML_LOW_MEMORY	Server application has outrun client, consuming large amounts of memory.
14011	DDEML_MEMORY_ERROR	A memory allocation failed.
14013	DDEML_NO_CONV_ESTABLISHED	Client conversation attempt failed.
14014	DDEML_POKEACTIMEOUT	Transaction failed.
14015	DDEML_POSTMSG_FAILED	Request for a synchronous poke transaction has timed out.
14016	DDEML_REENTRANCY	An application with a synchronous transaction in progress attempted to initiate another transaction, or a DDEML callback function called `DdeEnableCallback`.
14017	DDEML_SERVER_DIED	Server-side transaction attempted on conversation terminated by client, or service terminated before completing a transaction.
14018	DDEML_SYS_ERROR	Internal error in the DDMEML.
14019	DDEML_UNADVACKTIMEOUT	Request to end advise has timed out.
14020	DDEML_UNFOUND_QUEUE_ID	Invalid transaction identifier passed to DDEML function.

Appendix C

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the Technical Support Form before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(800) 433-3488 (toll-free U.S. and Canada)
Technical Support fax: (512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	03 879 9422	03 879 9179
Austria	0662 435986	0662 437010 19
Belgium	02 757 00 20	02 757 03 11
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 65 33 00	1 48 65 19 07
Germany	089 7 14 50 93	089 7 14 60 35
Italy	02 48301892	02 48301915
Japan	03 3788 1921	03 3788 1923
Netherlands	01720 45761	01720 42140
Norway	03 846866	03 846860
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 27 00 20	056 27 00 25
U.K.	0635 523545	0635 523154
	or 0800 289877 (in U.K. only)	

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ______________________________

Company ______________________________

Address ______________________________

____

Fax (___) ____________ Phone (___) ____________

Computer brand ____________ Model ____________ Processor ____________

Operating system (include version number) ______________________________

Clock speed ______MHz RAM _____M Display adapter ____________

Mouse ___yes ___no Other adapters installed ____________

Hard disk capacity _____M Brand ____________

Instruments used ______________________________

____

Boards installed (include revision and configuration) ____________

____

____

LabVIEW Version ____ VI Libraries installed (other than standard libraries) ____________

____

The problem is: ______________________________

____

____

List any error messages: ______________________________

____

The following steps reproduce the problem: ____________

____

____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **LabVIEW® for Windows User Manual**

Edition Date: **August 1993**

Part Number: **320534-01**

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name

Title

Company

Address

Phone (____)

Mail to:
Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to:
Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

Glossary

Glossary

Glossary

Glossary

Prefix	Meaning	Value
μ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}

Symbols

∞ Infinity.

π Pi.

A

A Amperes.

absolute path Relative file or directory path that describes the location relative to the top of level of the file system.

active window Window that is currently set to accept user input, usually the frontmost window. The title bar of an active window is highlighted. You make a window active by clicking on it, or by selecting it from the **Windows** menu.

ANSI American National Standards Institute.

array Ordered, indexed set of data elements of the same type.

array shell Front panel object that houses an array. It consists of an index display, a data object window, and an optional label. It can accept various data types.

artificial data dependency Condition in a dataflow programming language in which the arrival of data rather than its value triggers execution of a node.

ASCII American Standard Code for Information Interchange.

asynchronous execution Mode in which multiple processes share processor time. For example, one process executes while others wait for interrupts during device I/O or while waiting for a clock tick.

ATE Automatic test equipment.

auto-indexing

Capability of loop structures to disassemble and assemble arrays at their borders. As an array enters a loop with auto-indexing enabled, the loop automatically disassembles it with scalars extracted from one-dimensional arrays, one-dimensional arrays extracted from two-dimensional arrays, and so on. Loops assemble data into arrays as they exit the loop according to the reverse of the same procedure.

autoscaling

Ability of scales to adjust to the range of plotted values. On graph scales, this feature determines maximum and minimum scale values, as well.

autosizing

Automatic resizing of labels to accommodate text that you enter.

B

block diagram

Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the Diagram window of the VI.

BNF

Backus-Naur Form. A common representation for language grammars in computer science.

Boolean controls and indicators

Front panel objects used to manipulate and display or input and output Boolean (TRUE or FALSE) data. Several styles are available, such as switches, buttons and LEDs.

breakpoint

Mode that halts execution when a subVI is called. You set a breakpoint by clicking on the Breakpoint button in the execution palette.

broken VI

VI that cannot be compiled or run; signified by a broken arrow in the Run button.

Bundle node

Function that creates clusters from various types of elements.

byte stream file

File that stores data as a sequence of ASCII characters or bytes.

C

c

Speed of light.

case

One subdiagram of a Case structure.

Case structure

Conditional branching control structure, which executes one and only one of its subdiagrams based on its input. It is the combination of the IF, THEN, ELSE, and CASE statements in control flow languages.

cast

To change the type descriptor of a data element without altering the memory image of the data.

chart

See scope chart, strip chart, and sweep chart.

CIN

See Code Interface Node.

clone	To duplicate graphics by selecting and dragging while pressing a control key.
cloning	To make a copy of a control or some other LabView object by clicking the mouse button while pressing the <Ctrl> key and dragging the copy to its new location.
cluster	A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
cluster shell	Front panel object that contains the elements of a cluster.
Code Interface Node	Special block diagram node through which you can link conventional, text-based code to a VI.
coercion	The automatic conversion LabVIEW performs to change the numeric representation of a data element.
coercion dot	Glyph on a node or terminal indicating that the numeric representation of the data element changes at that point.
Coloring tool	Tool you use to color objects and backgrounds.
compile	Process that converts high-level code to machine-executable code. LabVIEW automatically compiles VIs before they run for the first time after creation or alteration.
conditional terminal	The terminal of a While Loop that contains a Boolean value that determines whether the VI performs another iteration.
connector	Part of the VI or function node that contains its input and output terminals, through which data passes to and from the node.
connector pane	Region in the upper right corner of a Panel window that displays the VI terminal pattern. 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.
Controls menu	Menu of controls and indicators.
conversion	Changing the type of a data element.
count terminal	The terminal of a For Loop whose value determines the number of times the For Loop executes its subdiagram.
CPU	Central processing unit.

current VI

VI whose Panel window, Diagram window, or icon editor window is the active window.

custom PICT controls and indicators

Controls and indicators whose parts can be replaced by graphics you supply.

D

data acquisition

Process of acquiring data, typically from A/D or digital input plug-in boards.

data dependency

Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *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 log data.

data storage formats

The arrangement and representation of data stored in memory.

data type descriptor

Code that identifies data types, used in data storage and representation.

datalog file

File that stores data as a sequence of records of a single, arbitrary data type that you specify when you create the file. While all the records in a datalog file must be of a single type, that type can be complex; for instance, you can specify that each record is a cluster containing a string, a number, and an array.

Description box

Online documentation for a LabVIEW object.

destination terminal

See sink terminal.

Diagram window

VI window that contains the block diagram code.

dialog box

An interactive screen with prompts in which you specify additional information needed to complete a command.

dimension

Size and structure attribute of an array.

drag

To drag the mouse cursor on the screen to select, move, copy, or delete objects.

DUT

Device under test.

E

e Electronic charge.

edit mode The mode in which you create or edit a VI.

empty array Array that has zero elements, but has a defined data type. For example, an array that has a numeric control in its data display window but has no defined values for any element is an empty numeric array.

EOF End-of-File. Character offset of the end of file relative to the beginning of the file (that is, the EOF is the size of the file).

execution highlighting Feature that animates VI execution to illustrate.the data flow in the VI.

external routine *See* shared external routine.

F

FFT Fast Fourier transform.

file refnum An identifier that LabVIEW associates with a file when you open it. You use the file refnum to specify that you want a function or VI to perform an operation on the open file.

flattened data Data of any type that has been converted to a string, usually, for writing it to a file.

For Loop Iterative loop structure that executes its subdiagram a set number of times. Equivalent to conventional code: For i=0 to n-1, do ....

Formula Node Node that executes formulas that you enter as text. Especially useful for lengthy formulas that would be cumbersome to build in block diagram form.

frame Subdiagram of a Sequence structure.

free label Label on the front panel or block diagram that does not belong to any other object.

front panel The interactive user interface of a VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.

function Built-in execution element, comparable to an operator, function, or statement in a conventional language.

G

G The LabVIEW graphical programming language.

g Gram.

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.

GPIB
General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.

graph control
Front panel object that displays data in a Cartesian plane.

H

handle
Pointer to a pointer to a block of memory; handles reference arrays and strings. An array of strings is a handle to a block of memory containing handles to strings.

Help window
Special window that displays the names and locations of the terminals for a function or subVI, the description of controls and indicators, the values of universal constants, and descriptions and data types of control attributes.

hex
Hexadecimal. A base-16 number system.

hierarchical menu
Mcnu that contains submenus or palettes.

housing
Nonmoving part of front panel controls and indicators that contains sliders and scales.

Hz
Hertz. Cycles per second.

I

icon
Graphical representation of a node on a block diagram.

icon editor
Interface similar to that of a paint program for creating VI icons.

icon pane
Region in the upper right corner of the Panel and Diagram windows that displays the VI icon.

IEEE
Institute for Electrical and Electronic Engineers

indicator
Front panel object that displays output.

Inf
Digital display value for a floating-point representation of infinity.

inplace execution
Ability of a function or VI to reuse memory instcad of allocating more.

instrument driver
VI that controls a programmable instrument.

I/O Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.

iteration terminal The terminal of a For Loop or While Loop that contains the current number of completed iterations.

J

J Joule. Absolute unit of work or energy equal to 10^7 ergs.

L

label Text object used to name or describe other objects or regions on the front panel or block diagram.

Labeling tool Tool used to create labels and enter text into text windows.

LabVIEW Laboratory Virtual Instrument Engineering Workbench.

LED Light-emitting diode.

legend Object owned by a chart or graph that display the names and plot styles of plots on that chart or graph.

M

MB Megabytes of memory.

marquee A moving, dashed border that surrounds selected objects.

matrix Two-dimensional array.

menu bar Horizontal bar that contains names of main menus.

modular programming Programming that uses interchangeable computer routines.

N

NaN Digital display value for a floating-point representation of *not a number*, typically the result of an undefined operation, such as log(-1).

nodes Execution elements of a block diagram consisting of functions, structures, and subVIs.

nondisplayable characters ASCII characters that cannot be displayed, such as newline, tab, and so on.

not-a-path A predefined value for the path control that means the path is invalid.

not-a-refnum

A predefined value that means the refnum is invalid.

numeric controls and indicators

Front panel objects used to manipulate and display or input and output numeric data.

O

object

Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures. Described in your tutorial manual.

Operating tool

Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

P

palette menu

Menu that displays a palette of pictures that represent possible options.

Panel window

VI window that contains the front panel, the execution palette and the icon/connector pane.

platform

Computer and operating system.

plot

A graphical representation of an array of data shown either on a graph or a chart.

polymorphism

Ability of a node to automatically adjust to data of different representation, type, or structure.

pop up

To call up a special menu by clicking (usually on an object) with the right mouse button.

pop-up menus

Menus accessed by popping up, usually on an object. Menu options pertain to that object specifically.

Positioning tool

Tool used to move and resize objects.

probe

Debugging feature for checking intermediate values in a VI.

programmatic printing

Automatic printing of a VI front panel after execution.

pseudo-code

Simplified language-independent representation of programming code.

pull-down menus

Menus accessed from a menu bar. Pull-down menu options are usually general in nature.

R

reentrant execution

Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.

refnum
A file refnum is an identifier of open files that can be referenced by other VIs. A conversation refnum is used to identify a DDE conversation.

representation
Subtype of the numeric data type, of which there are signed and unsigned byte, word, and long integers, as well as single-, double-, and extended-precision floating-point numbers, both real and complex.

resizing handles
Angled handles on the corner of objects that indicate resizing points.

ring control
Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

run mode
The mode in which you execute a VI.

S

scalar
Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans, strings, and clusters are explicitly singular instances of their respective data types.

scale
Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.

scope chart
Numeric indicator modeled on the operation of an oscilloscope.

sequence local
Terminal that passes data between the frames of a Sequence structure.

Sequence structure
Program control structure that executes its subdiagrams in numeric order. Commonly used to force nodes that are not data-dependent to execute in a desired order.

shared external routine
Subroutine that can be shared by several CIN code resources.

shift register
Optional mechanism in loop structures used to pass the value of a variable from one iteration of a loop to a subsequent iteration.

sink terminal
Terminal that absorbs data. Also called a destination terminal.

slider
Moveable part of slide controls and indicators.

source terminal
Terminal that emits data.

string controls and indicators
Front panel objects used to manipulate and display or input and output text.

strip chart
A numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.

structure
Program control element, such as a Sequence, Case, For Loop, or While Loop.

subdiagram
Block diagram within the border of a structure.

subVI
VI used in the block diagram of another VI; comparable to a subroutine.

sweep chart	Similar to scope chart; except a line sweeps across the display to separate old data from new data.

T

table-driven execution	A method of execution in which individual tasks are separate cases in a Case structure that is embedded in a While Loop. Sequences are specified as arrays of case numbers.
terminal	Object or region on a node through which data passes.
tool	Special LabVIEW cursor you can use to perform specific operations.
top-level VI	VI at the top of the VI hierarchy. This term distinguishes the VI from its subVIs.
tunnel	Data entry or exit terminal on a structure.
type descriptor	*See* data type descriptor.

U

universal constant	Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, for example, pi.
user-defined constant	Block diagram object that emits a value you set.
UUT	Unit under test.

V

V	Volts.
VI	*See* virtual instrument.
VI library	Special file that contains a collection of related VIs for a specific use. See the *VI Libraries* section of Chapter 1, *Introduction to LabVIEW*, for more information.
virtual instrument	LabVIEW program; so called because it models the appearance and function of a physical instrument.

W

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.
wire	Data path between nodes.
Wiring tool	Tool used to define data paths between source and sink terminals.

Index

Index

Index

Index

This manual represents a commitment from National Instruments to the environment. The paper used in this manual is made from a large percentage of recycled material. In the future, National Instruments will look for new and better ways to further our commitment to the environment.